Fish Engine Simulator Documentation

by Abdelkader KANTAOUI

First Release: 15th August 2023

Documentation Summary: check out https://perso.limos.fr/ abkantaoui/MyWork/FishEngine/index.html.

This PDF serves as documentation for the Fish Engine Simulator. It provides comprehensive information about the simulator's features, usage, and implementation details.

Author's Portfolio:

For more information about my work, please visit my portfolio at perso.limos.fr/ abkantaoui.

Contents

1 User Guide

1.1 Introduction

Welcome to the Fish Simulator - Dive into an Imaginary Underwater Realm

Welcome to the Fish Simulator, an enthralling open-source project that invites you to explore an imaginative underwater world. This simulator goes beyond mere animation; it provides you with an intricate representation of movement for each instance, be it fish or predator. At the heart of the simulation lies the `aquaengine.c` file, where the main mechanisms for intelligent movements and interactions are implemented.

Engaging Movements: In this aquatic realm, the simulator breathes life into every instance by employing specific main mechanisms of movement. You will witness fish elegantly swimming together, forming groups that flow harmoniously around uncommon barycenters. Predators, embodying the "High risk, high profit" strategy, deftly navigate the waters in pursuit of areas with concentrated fish populations.

Dynamic Interactions: Within this mesmerizing underwater ecosystem, instances interact dynamically with one another, crafting an ever-changing environment. The behavior of fish is shaped by their interactions with other fish and predators, fostering an intelligent and cohesive community. These interactions, orchestrated by the `aquaengine.c` file's realm-specific probabilistic function, lead to the emergence of awe-inspiring patterns and behaviors.

Realm Mechanism Variations: The simulator's imaginary world thrives on variation. You can experiment with a range of settings and parameters, witnessing the profound impact of each alteration on fish, predators, and the aquatic environment. As you navigate this immersive realm, you will uncover the delicate balance that governs the behaviors of its inhabitants.

Requirements: To fully immerse yourself in the intricacies of the Fish Simulator and understand the underlying mechanisms in `aquaengine.c`, you are encouraged to possess a good understanding of trigonometry and mathematical functions. These fundamental concepts are integral to comprehending the intelligent movements and interactions that breathe life into this vibrant underwater world.

This User Guide embarks on a journey through the enchanting Fish Simulator, with a focus on `aquaengine.c` and the core elements that shape its existence. As you dive deeper into the following chapters, a world of intelligent fish movements, predator-prey dynamics, and the realm's ever-changing behavior will unfold before you.

With this introduction, you are invited to delve into the fascinating world we've created, centered around the `aquaengine.c` file, igniting your curiosity to explore the intricacies of the Fish Simulator. Tailor the content further to match your specific goals and engage with this captivating underwater realm. If you need any more assistance or have other questions, feel free to ask!

The 'User Guide' serves as the **core resource** for **researchers, enthusiasts, and mathematicians** who seek to delve into the inner workings of the **AquaEngine simulator**. With an emphasis on **mathematical models** and **customizable components**, the **User Guide** stands as the **primary focal point** for those eager to explore the intricacies of aquatic simulations. As a **hub of modifiable aspects**, including **mathematical algorithms** and **physics simulations**, it **empowers users** to **harness the potential** of **AquaEngine** and **unleash their creativity** in **modeling and experimenting** with various aquatic behaviors. Whether you're an **expert mathematician** or an **eager learner**, the **User Guide** is your **gateway to understanding, experimenting, and pushing the boundaries** of this aquatic world simulation.

1.2 Header File (`aquaengine.h`)

Overview

`aquaengine.h` serves as the header file for the Fish Simulator's engine. It contains essential definitions, data structures, and function prototypes that drive the simulation. The engine is designed to provide intelligent movement for fish and predator instances, interaction management, and core functionalities to create a captivating underwater realm.

Graphic Mode for the Engine

The Fish Simulator's engine offers a versatile graphic mode that can be enabled by defining `GRAPHIC_MODE_FOR_THE_ENGINE`. This graphic mode utilizes the SDL2 library for graphics and animation, providing a visually appealing underwater world. However, if you wish to use OpenGL instead of SDL2, transitioning is relatively easy. Simply track occurrences of `GRAPHIC_MODE_FOR_TH` in the code and replace any SDL-specific code with its equivalent in OpenGL.

Please note that OpenGL usage will require appropriate OpenGL setup, rendering, and shader management. Transitioning to OpenGL allows for more advanced visual effects and performance optimizations.

Constants and Definitions

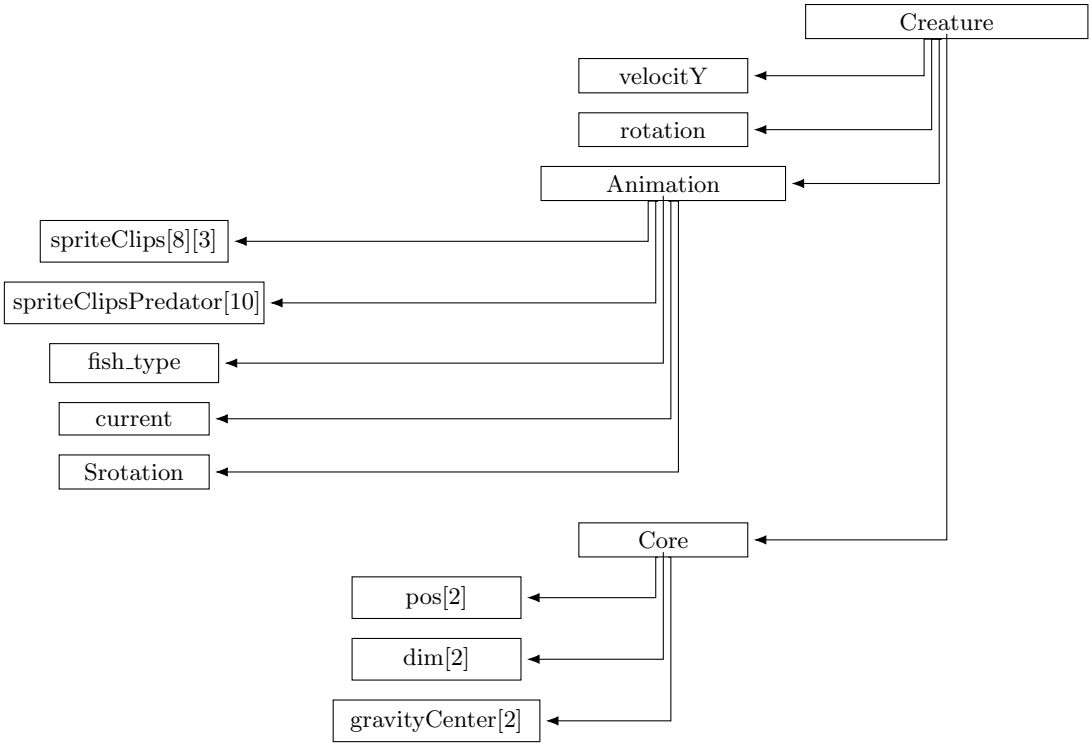
The following constants and definitions are important for the functionality of the engine:

- `FSPRITE_WIDTH` and `FSPRITE_HEIGHT`: Width and height of individual fish sprites for animation.
- `PSPRITE_WIDTH` and `PSPRITE_HEIGHT`: Width and height of predator sprites for animation.
- `FNUM_SPRITES_PER_ROW` and `FNUM_SPRITES_PER_COL`: Number of fish sprites per row and column in the sprite sheet.
- `NOMBRE_SPRITES`: Total number of different fish sprites available for animation.
- `FishHeadBoxW` and `FishHeadBoxH`: Width and height of the fish's head hitbox for collision detection.
- `PredatorHeadBoxW` and `PredatorHeadBoxH`: Width and height of the predator's head hitbox for collision detection.
- `SAFERADIUS`: Safety radius for fish; distance at which they start avoiding collisions with other fish.
- `CRITIQUERADIUS`: Critical radius for fish; distance at which they start to exhibit critical behavior.
- `WINDOW[2]`: Array representing the window dimensions (width and height) for the simulator's display.

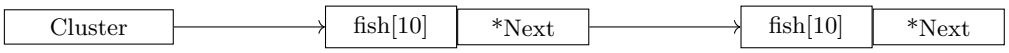
Data Structures

The header file defines the following data structures:

- `Animation`: Structure representing animation properties for fish instances. Includes sprite clips, fish type, current animation frame, and rotation.
- `Core`: Structure holding core properties of a creature (fish or predator). Includes position, dimensions, and gravity center information.
- `Creature`: Structure representing individual creatures (fish or predator). Contains core properties, velocity, and rotation. In graphic mode, it also includes animation properties.



- `Cluster`: Structure representing a linked list of fish instances, grouped together for movement management. This structure efficiently manages a collection of fish instances within a linked list format. Each `Cluster` contains an array of `Creature` elements, allowing it to hold up to 10 fish instances. Additionally, each `Cluster` maintains a pointer to the next cluster in the list, if available. This design offers an important trade-off between memory usage and performance, as it optimizes memory allocation by grouping a limited number of fish instances (up to 10) while enabling efficient traversal through the linked list. The array-based approach enhances data locality within the list, minimizing cache misses and facilitating faster access to neighboring fish instances for movement calculations. This design consideration ensures that the `Cluster` structure strikes a balance between memory efficiency and movement management performance, making it a key component of the Fish Simulator's engine.



Function Prototypes

1. `void Initialize_fish(Creature* fishy, Creature* predator);` Initialize a fish instance with given parameters.
2. `void Eaten(Creature* creature);` Handle the removal of a creature that has been eaten. (Note: This function is currently disabled and can be enabled by uncommenting the inside code in the `.c` file.)
3. `void Initialize_Predator(Creature* predator, Creature* TheOtherPredator);` Initialize a predator instance with given parameters.
4. `bool IsRectEmpty(Core core);` Check if a rectangle defined by its core is empty (no creature present).
5. `void Initialize_Cluster(Cluster* cluster, Creature* predator);` Initialize a cluster of fish instances with a predator instance.
6. `void Free_Creature(Creature* fishy);` Free memory allocated for a creature instance.
7. `void Free_Cluster(Cluster** Clusty);` Free memory allocated for a cluster of fish instances.
8. `double* getFictifCoordinates(double pivot, double guest, int axis_dimension);` Get fictitious coordinates for a creature's pivot and guest.
9. `double Folded_Space_Distance(double x1, double y1, double x2, double y2);` Calculate the distance in a folded space between two creatures.

10. `double Customized_Selection(int k, double l);` Perform a customized selection based on a given parameter and evaluation function.

11. `double Standard_Evaluation(int k, double l);` Evaluate a parameter using a standard evaluation function.

12. `void Centrer_Gravity(Creature* creature, Cluster* cluster, double* t_X, double* t_Y, double (*Evaluate)(int double));` Determine the gravity center of a creature based on a cluster and evaluation function. The function pointer `Evaluate` influences the center of gravity calculation.

13. `double Critique_Probability(Creature* creature, Creature* predator);` Calculate the probability of critical behavior for a creature in relation to a predator.

14. `void Revolving_Around(Creature* creature, double disk_radius, double CenterX, double CenterY, Creature* predator);` Make a creature revolve around a given point (disk) with a specific radius. The function modifies the rotation and velocity module.

15. `void Seeking_Around(Creature* creature, double CenterX, double CenterY, Creature* predator);` Make a creature seek a given point. The function modifies the rotation.

16. `void mv_object(Cluster* cluster, Creature* creature, Creature* predator, bool Is_Predator);` Move a cluster of fish and predator instances based on their interactions. The boolean `Is_Predator` is a security measure to identify if `creature` is a predator.

1.3 Movement Functions:

mv_object

Purpose:
The `mv_object` function is a crucial movement function within the Fish Simulator's engine. It handles the movement of individual creatures (fish or predator) within the underwater realm. The function calculates the new position, rotation, and velocity of the creature based on various factors, including window boundaries, interaction with other creatures, and intelligent behaviors specific to each creature type.

Inputs:

- `cluster`: Pointer to the cluster of fish instances that the creature belongs to. This allows for interactions and movement management within the group.
- `creature`: Pointer to the creature instance for which movement is being calculated.
- `predator`: Pointer to the predator instance (if applicable). This is used for predator-specific behavior.
- `Is_Predator`: A boolean flag that serves as a security measure to determine whether the creature is a predator or a fish instance.

Outputs:
None. The function updates the position, orientation (rotation), and velocity of the creature.

Details:

1. The `mv_object` function starts by checking if the creature's core rectangle is empty (the creature no longer exists). If so, it does not perform any movement.
2. Next, the function checks if the creature has crossed the window boundaries. If so, it implements periodic boundary conditions, ensuring that the creature appears on the opposite side of the window. This creates a toroidal (wrap-around) effect, simulating the underwater realm as a continuous environment.
3. The function then calculates the gravity center for the creature based on its interaction with the cluster and its neighbors. The `Centrer_Gravity` function is called to determine the average position of nearby creatures, affecting the movement of the current creature.

4. Depending on whether the creature is a fish or a predator (`Is_Predator` flag), the function proceeds with different movement strategies:
    - For fish instances (`Is_Predator = false`), the `Revolving_Around` function is called. This function makes the fish revolve around the calculated gravity center, promoting cohesive group behavior. The `Revolving_Around` function modifies the creature's rotation and velocity module accordingly.
    - For predator instances (`Is_Predator = true`), the `Seeking_Around` function is called. This function makes the predator seek the calculated gravity center, aiming to approach fish groups. The `Seeking_Around` function modifies the creature's rotation and velocity module accordingly.
5. Finally, the function calculates the new velocity (`VelX` and `VelY`) for the creature based on its speed and current rotation. It updates the creature's position by adding the velocity components.

Note:

The `(false || ..)` check serves a purpose for stopping the creature's movement. However, the current implementation has it as false, allowing the creature to continue moving. If you want to stop all movement, you can change it to true, and the movement will be halted.

Revolving Around

The `Revolving_Around` function is a crucial component of the Fish Simulator's engine, responsible for simulating the revolving movement of a fish instance around a specified point in the underwater realm. This function calculates the new orientation (rotation) and velocity of the fish, enabling it to navigate around the given center in a cohesive manner.

Function Signature:

```
void Revolving_Around(Creature *creature, double disk_radius,
                      double CenterX, double CenterY, Creature *predator);
```

Parameters:

- `creature`: A pointer to the fish instance for which the revolving movement is being calculated.
- `disk_radius`: The radius of the disk (circular area) around the center (`CenterX` and `CenterY`) where the fish revolves.
- `CenterX`: The X-coordinate of the center of the disk.
- `CenterY`: The Y-coordinate of the center of the disk.
- `predator`: A pointer to the predator instance. This is used to determine the probability of critical behavior for the fish.

Outputs:

None. The function updates the rotation (`creature->rotation`) and velocity (`creature->velocityY`) of the fish instance.

Details:

- The distance (`dist`) between the fish and the center of the revolving disk is calculated using the Euclidean distance formula.
- **Revolving Theta and Smooth Rotation:** $\text{AntiCollapse} = \frac{1}{(x+2^{+\frac{1}{n}})^n}$ $\text{Collapse} = A^{-x}$

  The function plays a pivotal role in determining the angle of rotation (θ_{revolve}) by which the fish revolves around the specified center. This angle is not simply a fixed value, but rather a dynamic quantity influenced by multiple factors.

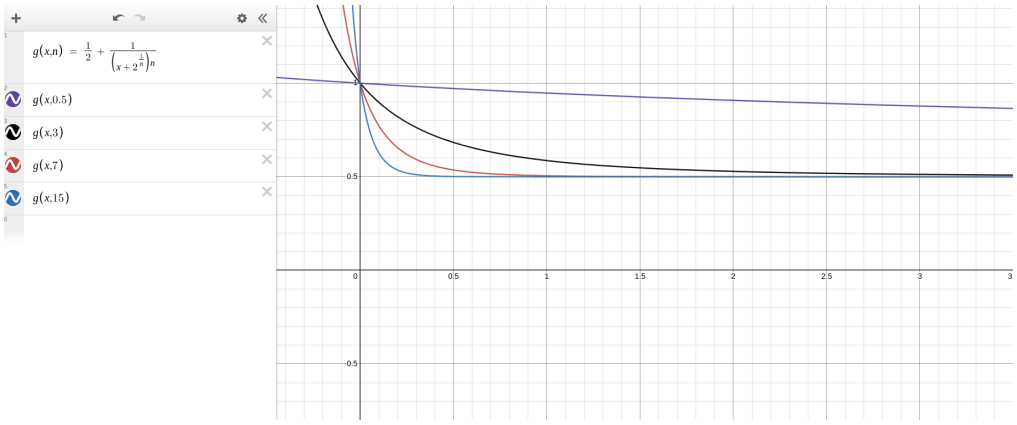


Figure 1: AntiCollapse(x) with differents n values

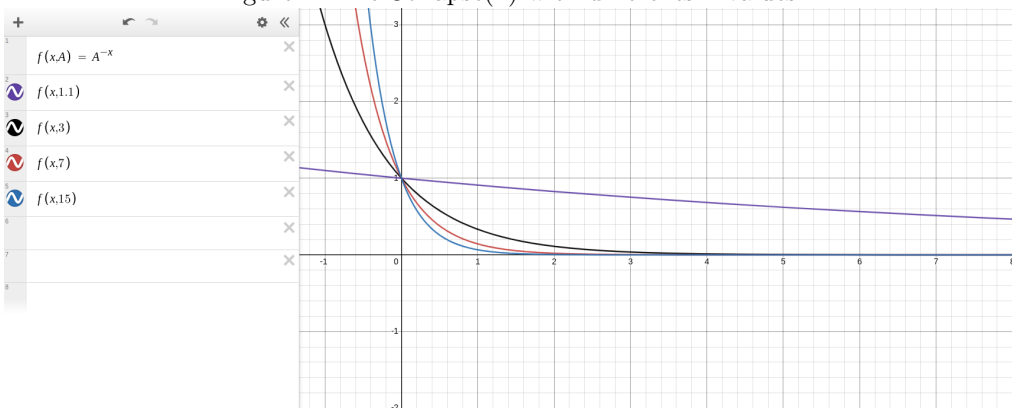


Figure 2: Collapse(x) with differents A values

A key element in the formula for θ_{revolve} is the term $\frac{1}{(x+2^{+\frac{1}{n}})^n}$, where x is the normalized distance between the fish and the center, and n is a parameter that modulates the sensitivity of the collapsing effect.

The function $\frac{1}{(x+2^{+\frac{1}{n}})^n}$ exhibits intriguing behavior. As x approaches infinity (indicating that the fish is at a significant distance from the center), the term approaches zero. In this scenario, θ_{revolve} tends towards 0,because `AntiCollapse` tends to $\frac{1}{2}$, and `Collapse` tend to 0, resulting in a less pronounced revolving behavior. The fish becomes less influenced by the center as it moves further away, resulting in a smoother and more linear movement pattern.

Conversely, as x approaches zero (signifying that the fish is very close to the center), the term $\frac{1}{(x+2^{+\frac{1}{n}})^n}$ increases. Consequently, θ_{revolve} becomes larger, leading to a more significant revolving effect. When the fish is in proximity to the center, the revolving behavior becomes more dominant, allowing the fish to navigate precisely around the central point. This dynamic adjustment of θ_{revolve} ensures that the fish's rotation smoothly transitions from 0 to $\frac{\pi}{2}$ (90 degrees) as x ranges from $+\infty$ to 0^+.

By elegantly blending the influences of x, n, and other parameters, AquaEngine creates a realistic and captivating representation of revolving behavior. The gradual shift from subtle rotation to pronounced swirling as the fish moves from near and far distances from the center contributes to the overall authenticity and immersion of the underwater simulation.

(See fig.1 and fig.2)

- **Influential Parameters and Collapsing Effect** Where n is a parameter that influences the sensitivity of the collapsing effect. As n increases, the collapsing effect becomes more pronounced, causing the fish to rotate more rapidly and appear tightly clustered around the center. Conversely, smaller values of n lessen the collapsing effect, allowing the fish to maintain a more stable and uniform rotation, and preventing the formation of a singularity at the center.

  To elaborate further, consider the interplay between the parameters A and n as a delicate balance between forces. `The collapsing factor` can be envisioned as analogous to the gravitational forces in a celestial system. `The anti-collapsing factor`, on the other hand, acts like the nuclear fusion process within a star, providing the necessary pressure to counteract gravitational collapse.

  This intricate dance between A and n is what ultimately shapes the fish's revolving behavior. As n adjusts the sensitivity

of this balance, it dictates whether the fish clusters intensely around the center or spreads out more evenly, akin to the gravitational pull of celestial bodies and the opposing forces that prevent their collapse.

In the context of AquaEngine's fish simulation, this dynamic equilibrium ensures that the fish movement remains visually appealing and natural, avoiding unrealistic singularities while capturing the essence of fluid motion. The manipulation of these parameters allows developers to finely craft the fish's behavior, striking the perfect balance between stability and motion.

This intricate interplay of forces, reminiscent of celestial dynamics, is a testament to the depth and complexity embedded within AquaEngine's simulation, showcasing the harmonious blend of mathematical principles and creative design that brings the underwater realm to life.

Samples in low debugmode
For $A = 1.01$ `and` $n = 0.01$, you can watch the simulation video at:
`https://youtu.be/_KlhxVlZZPA` (Well balanced, default)

For $A = 1.000001$ `and` $n = 0.01$, you can watch the simulation video at:
`https://youtu.be/oo6Tx6rrj8w` (Well balanced, more natural)

For $A = 1.01$ `and` $n = 1.5$, you can watch the simulation video at:
`https://youtu.be/OaCWuRtZ9yI` (Anti-Collapsing is weak, result in collapsing , permanent stability)

For $A = 100$ `and` $n = 0.01$, you can watch the simulation video at:
`https://youtu.be/nGWN_P74T7o` (Collapsing is weak, result in no revolving/spinning , permanent stability)

Undefined behavior Samples in low debugmode

`Note :` when A is less than 1, or n is negative, the behavior of the cluster is pretty undefined, but also very satisfying.
For $A = 0.69$ `and` $n = 0.001$, you can watch the simulation video at:
`https://youtu.be/YzmVAJjxqMU` (Collapsing is weak, result in no revolving/spinning , permanent stability)

- Sweet Radius and Rotation Behavior:
  - Repulsion occurs when the fish is too close to the sweet radius, preventing clustering and adding dispersion to the fish's movement.
  - Attraction guides the fish back towards the center of gravity when it is too far from the sweet radius, ensuring cohesive and organized flow.
- The probability of critical behavior for the fish is calculated using the `Critique_Probability` function, making the fish more cautious when predators are nearby.
- The `forwardAngle` represents the fish's direction relative to the center.
- The `target_angle` is a combination of `forwardAngle` and `Revolving_theta`, introducing randomness to the fish's movement.
- The **EscapeAngle** array is a crucial element within the `Revolving_Around` function. It serves as a means to adjust the rotation of a fish instance based on the probability of encountering predators. Comprised of two values, `EscapeAngle` is populated using the `Critique_Probability` function, which evaluates the potential threat posed by predators. These calculated values are then incorporated into the overall rotation computation, allowing the fish to make informed adjustments to its movement pattern. In this way, `EscapeAngle` contributes to the fish's ability to navigate its surroundings intelligently, enhancing the realism and dynamic nature of the underwater simulation.

  `Showcase with null target_angle` `https://youtu.be/YzmVAJjxqMU`
  `Showcase with default (A,n)` `https://youtu.be/HsKjOWGXxtA`
- The rotation (`creature->rotation`) is updated based on `target_angle` and the probability of critical behavior, accounting for caution in the presence of predators.
- The fish's velocity (`creature->velocityY`) is adjusted according to its distance from the center, influencing its speed during revolving.
- If graphic mode (`GRAPHIC_MODE_FOR_THE_ENGINE`) is enabled, the sprite rotation (`creature->anime.Srotation`) is updated for visual representation.

Alternative:

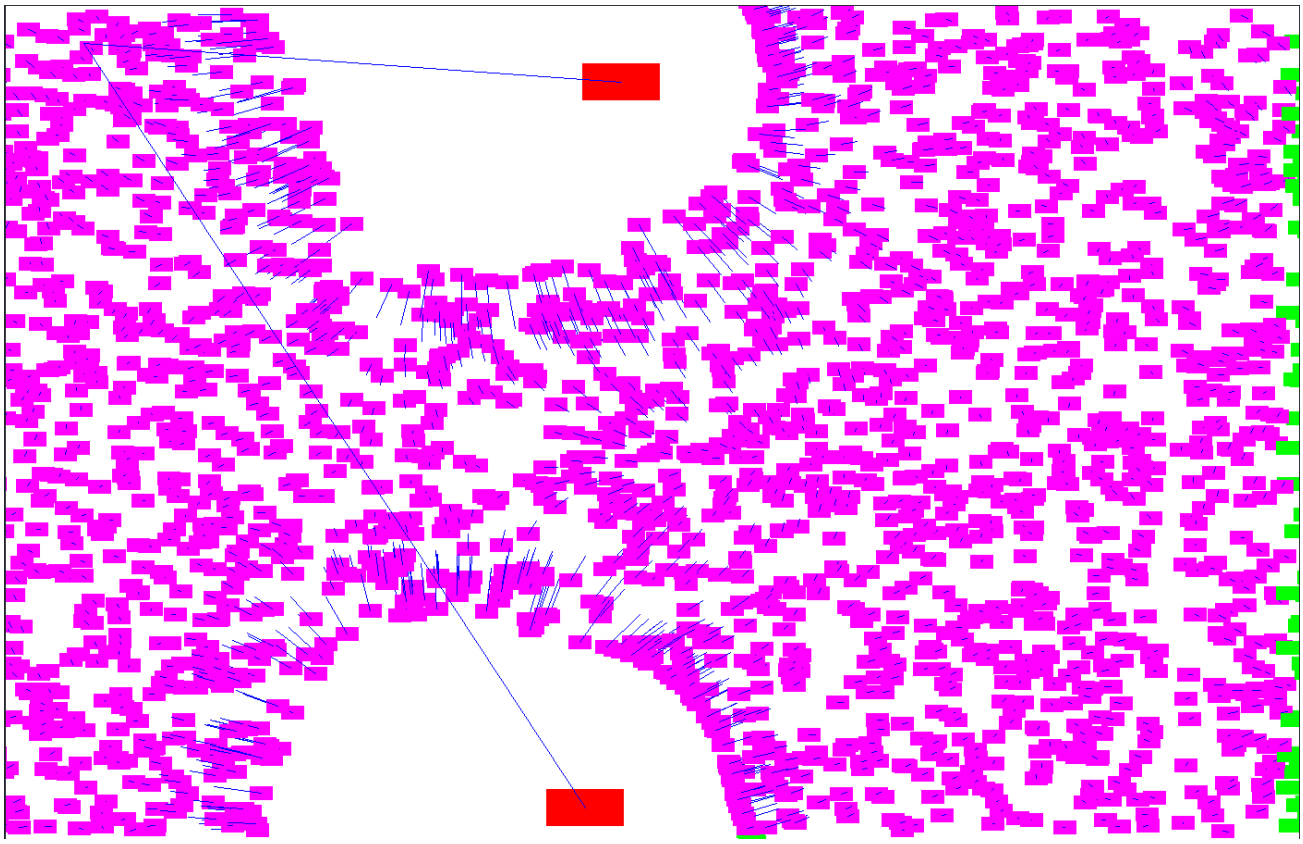


Figure 3: EscapeAngle

In delving into the mechanics of the `Revolving_Around` function, a captivating interplay between the parameters A and n becomes evident, particularly in their interaction with the Collapsing function (`Collapsing = pow(A, - x)`). This juxtaposition serves as a striking reminder of the profound symbiosis between mathematical abstraction and the observable world. To dissect this interaction and disentangle the influences of AntiCollapsing and Collapsing, let's Add $\frac{1}{2}$ to AntiCollapse so the formula become AntiCollapse $= \frac{1}{2} + \frac{1}{(x+2^{+\frac{1}{n}})^n}$.

As we embark on the journey to formulate a more nuanced expression for Collapsing, we traverse the realm of probability distributions with a creative twist. Drawing inspiration from the concept of normal distributions, which elegantly capture the clustering tendencies observed in various natural phenomena, we endeavor to adapt and refine this notion for `AquaEngine`'s fish simulation. Our aim is to modulate the clustering behavior while adhering to the inherent boundaries posed by the disk's circumference.

To achieve this, we introduce a modified function: Collapsing(x) $= e^{-\frac{|x-l|^s}{m}}$, where 'x' symbolizes the normalized distance, 'l' represents a parameter that determines the center of concentration, 's' controls the rate of decline, and 'm' ensures a smooth transition. While reminiscent of the familiar bell-shaped curve of a normal distribution, this function embraces a creative divergence to better suit the simulation's requirements.

The derivation of this formula from the classical normal distribution is an exercise in boundary conditions. We start by anchoring the endpoints, ensuring that Collapsing(0) and Collapsing(1) both equal a constant 'p'. This constraint elegantly imposes the necessary continuity and symmetry, resulting in the final expression: Collapsing $= p^{|2x-1|^s}$. This transformation offers a unique vantage point, allowing us to explore the intricate dynamics of clustering and dispersion around the central point.

In the context of `AquaEngine`, the introduction of the 's' parameter takes center stage, dictating the degree of clustering and distribution of fishes around the circular path. A larger 's' value fosters a more pronounced concentration, akin to tighter clustering, while a lower 's' value engenders a broader dispersion, preventing undue aggregation. This meticulous calibration empowers developers to sculpt a captivating spectrum of behaviors, encapsulating both focused cohesiveness and harmonious diffusion within the aquatic realm.

Samples in low debugmode

For $s = 16$ `and` $l = 2/3$ `and` $n = 20$ `and` $disk\_radius = 50$, you can watch the simulation video at:

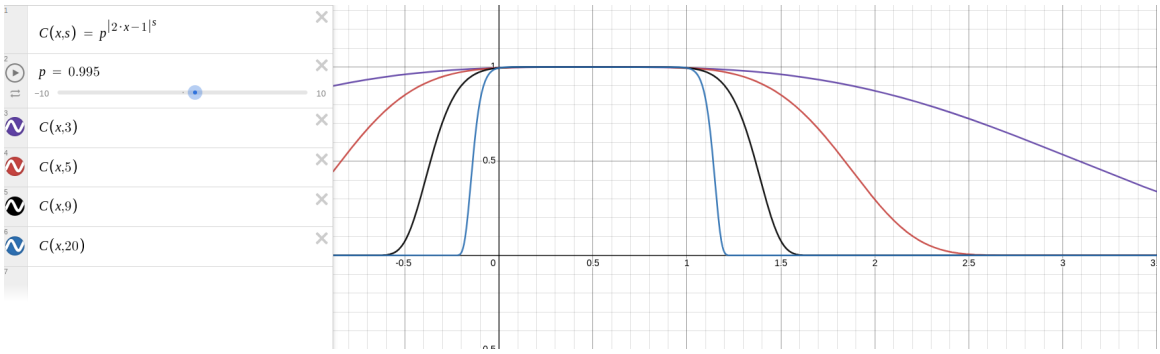


Figure 4: Alternative form for Collapse(x) with differents s values

Uncommun barycentre `https://youtu.be/FIZN7JVb4w0`
commun barycentre `https://youtu.be/gHI6bqQKqVg`

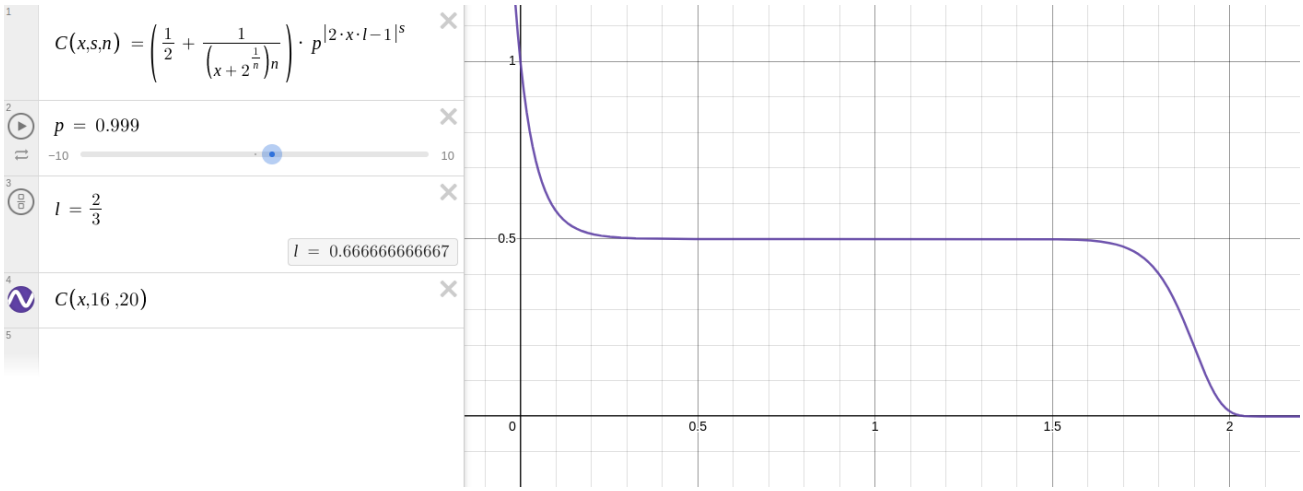


Figure 5: $s = 16$ and $l = 2/3$ and $n = 20$ and $disk\_radius = 50$

For $s = 3$ `and` $l = 7/3$ `and` $n = 7$ `and` $disk\_radius = 50$, you can watch the simulation video at:
Uncommun barycentre `https://youtu.be/nP08EFaCyEE`
commun barycentre `https://youtu.be/1aiLtmEawhw`

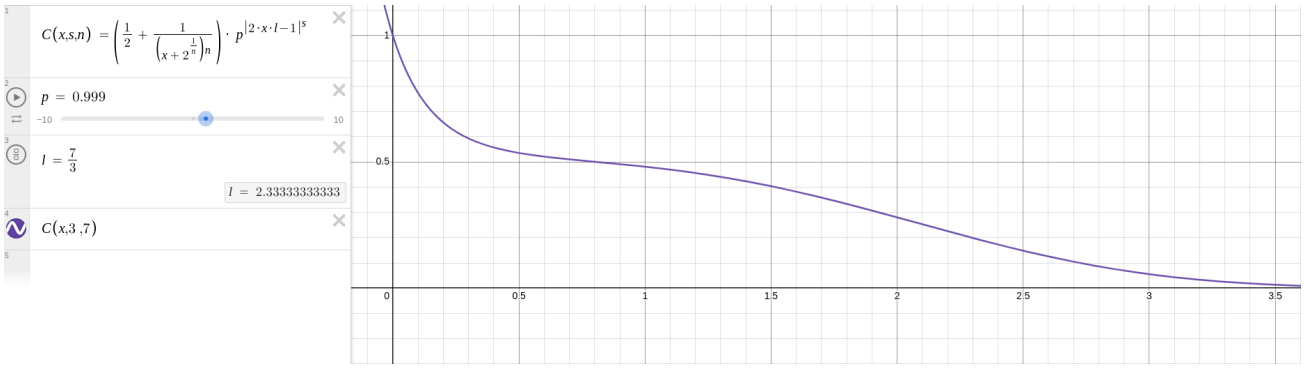


Figure 6: $s = 3$ and $l = 7/3$ and $n = 7$ and $disk\_radius = 50$

`Note:` In the last example, an intriguing property emerges: a lone intersection occurs at $y = \frac{\pi}{2}$. This singular point of reflection ensures that in cases where a cluster forms without significant distribution, it will gradually converge to a spherical configuration with a distinct radius. Investigate it further in the next video `https://youtu.be/8IstDDOrhkA`

Seeking_Around Function

The `Seeking_Around` function is a critical component of the fish simulation, responsible for simulating the seeking behavior of predator entities towards their prey. This function guides the predator's movement towards the center of mass of the prey, creating dynamic and engaging predator-prey interactions within the simulation.

Function Signature:

```
\noindent void Seeking_Around(Creature *creature, double CenterX, double CenterY, Creature *predator);
```

Parameters:

- **creature**: A pointer to the creature entity (predator) whose rotation needs to be adjusted based on seeking behavior.
- **CenterX**: The X-coordinate of the center of mass of the prey (target entity).
- **CenterY**: The Y-coordinate of the center of mass of the prey (target entity).
- **predator**: A pointer to the target entity (prey) that the predator is seeking.

Outputs:

None. The function updates the rotation (`creature->rotation`) and velocity (`creature->velocityY`) of the predator instance.

Explanation:

The `Seeking_Around` function is specifically designed for predator entities in the simulation. When called, this function calculates the distance between the predator (`creature`) and the prey (`predator`) using the `Folded_Space_Distance` function. Based on this distance, the function determines the seeking behavior of the predator towards its prey.

High Risk, High Profit Behavior:

In the context of the predator's seeking behavior, the concept of "high risk, high profit" refers to the predator's strategy when deciding whether to pursue the prey. The predator weighs the potential risks and rewards of pursuing the prey based on the proximity of the prey and the potential benefits of catching it.

When the prey is within a certain critical distance (`CRITIQUERADIUS`), the predator perceives a high chance of capturing the prey and obtaining a substantial reward (high profit). In this scenario, the predator intensifies its seeking behavior, pursuing the prey more aggressively.

However, as the distance between the predator and the prey increases, the probability of successfully capturing the prey decreases, and the potential reward diminishes (high risk). At a certain safe distance (`SAFERADIUS`), the predator decides to avoid excessive risks and becomes less aggressive in its pursuit of the prey.

This dynamic seeking strategy ensures that the predator adapts its behavior based on the perceived risk and potential reward of catching the prey, leading to more realistic and engaging predator-prey interactions in the simulation.

Mathematical Considerations:

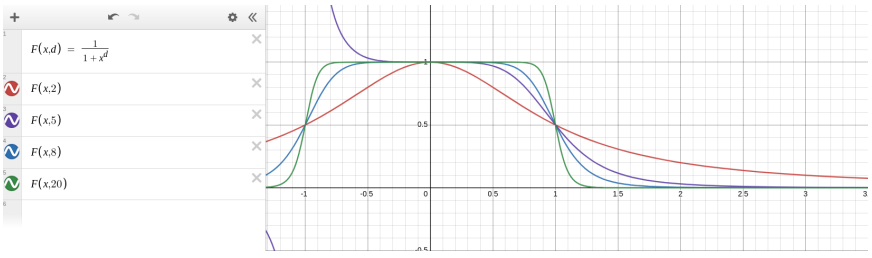


Figure 7: Diverging_probability(x) with differents div values

The mathematical function used to calculate the probability factor (`Diverging_probability`) in the `Seeking_Around` function plays a crucial role in avoiding the predator inter collision. As described earlier, the probability factor is influenced by the distance between the predators.

The function creates a smooth transition in the predator's seeking behavior, allowing the predator to closely track the prey when it is within a certain range (high profit) and pursue it more subtly as the distance increases (high risk), with ensuring a minimal distance far from the other predator.

Note:

- The `Seeking_Around` function, with its "high risk, high profit" behavior, ensures that the predators in the simulation demonstrate dynamic and strategic seeking behavior, resulting in engaging interactions with the prey.
- The combination of mathematical functions and strategic decision-making by the predators based on risk and reward enhances the realism and complexity of the predator-prey interactions in the fish simulation.
- It is important to note that the calculation of the center of gravity (`Center_gravity`) for fish and predators is different in the `mv_object` function. This distinction is made possible by using the `Operation` function pointer, allowing each entity type to have its unique behavior for center of gravity calculation.

Samples in low debugmode
Fist example, showcasing anti collision `https://youtu.be/M87AAOUkhps`
Second example `https://youtu.be/tjoHcVdCvug`

Critique_Probability

Calculate the probability of a creature being noticed by a predator based on certain parameters.

Function Signature:

```
double Critique_Probability(Creature *creature, Creature *predator, double p, double l, double s,
                                                    double B, double *EscapeAngle);
```

- ***creature:** A pointer to the creature for which the probability is being calculated.
- ***predator:** A pointer to the predator creature.
- **p,l,s,B** : A parameter affecting the probability calculation.
- ***EscapeAngle:** A pointer to a variable that will store the escape angle for the creature.

Outputs:

double: The calculated probability of the creature being noticed by the predator.

Purpose:

This function calculates the probability of a given creature being noticed by a predator based on the distance between them and several parameters that affect the probability calculation. The function uses mathematical formulas to model different aspects of the predator-prey interaction. It also contributes to creating a pulsing effect between predators in the simulation.

Insights into the Formulas:

1. The `Folded_Space_Distance` function calculates the distance x between the creature and the predator, determining their interaction.
2. Two probability calculation formulas are used based on the value of x compared to thresholds a and b:

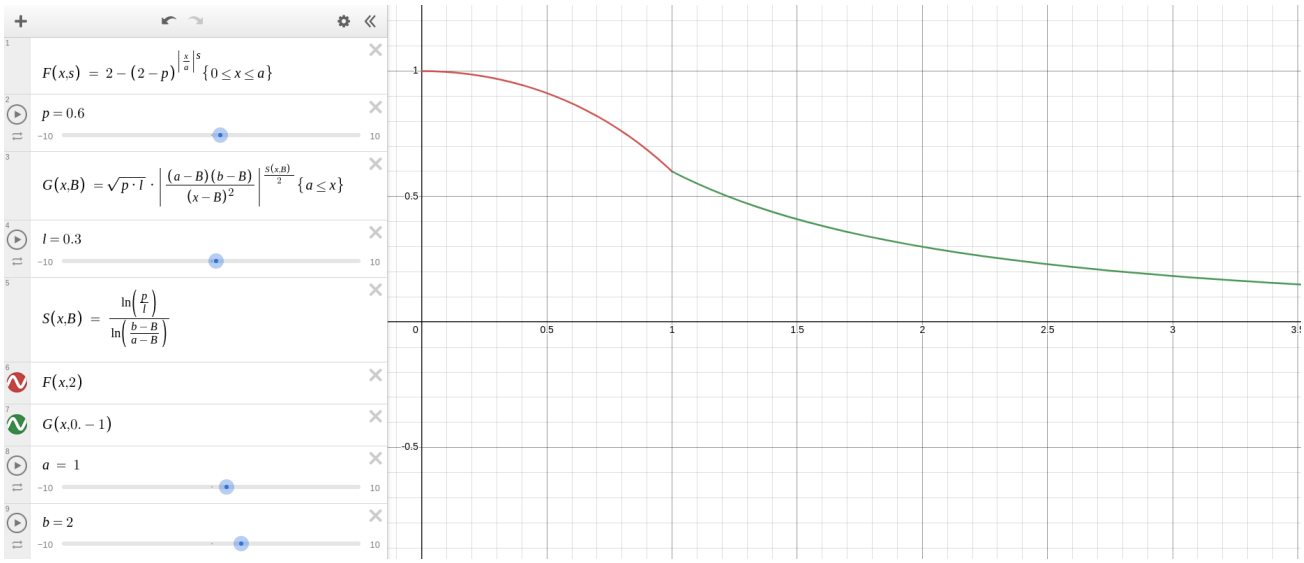


Figure 8: An overlook of the probabilistic function.

- If $x < a$, the F function formula is applied, representing the probability of predator detection within a certain radius.
- If $x \geq a$, the G function formula is used, representing the probability when the creature is outside the a radius.

3. F Function Formula:
   - Utilized when the creature is close to the predator ($x < a$).
   - Involves the p parameter, affecting the probability decrease as x approaches a.
   - The s parameter influences how the probability changes from a to 0. it's responsible for the curvature.
   - The formula $2-(2-p)^{(\frac{x}{a})^s}$ models decreasing probability as x nears a.
   - The original Model is $2-e^{A\cdot x^s}$ with border conditions F(0) = 1 and F(a) = p

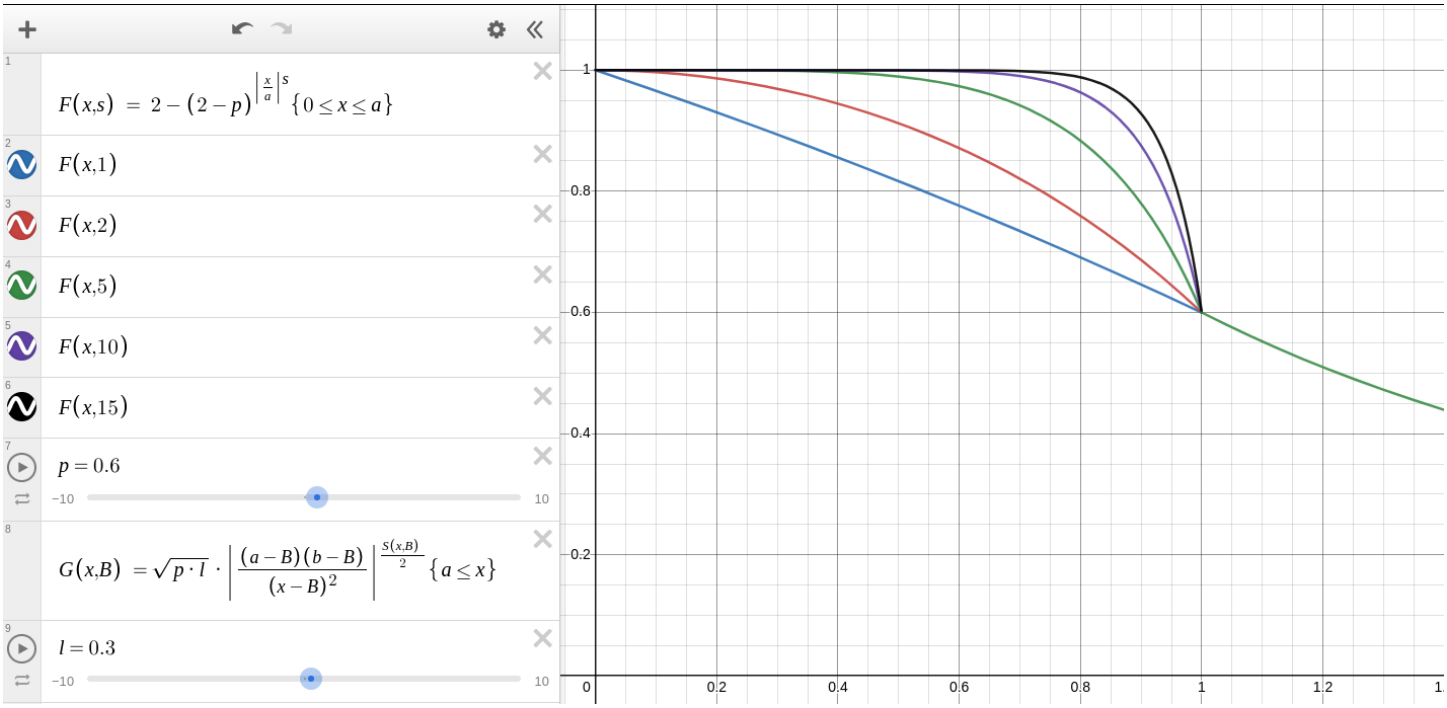


Figure 9: F(x) with different s values and for p= 0.6

4. G Function Formula:

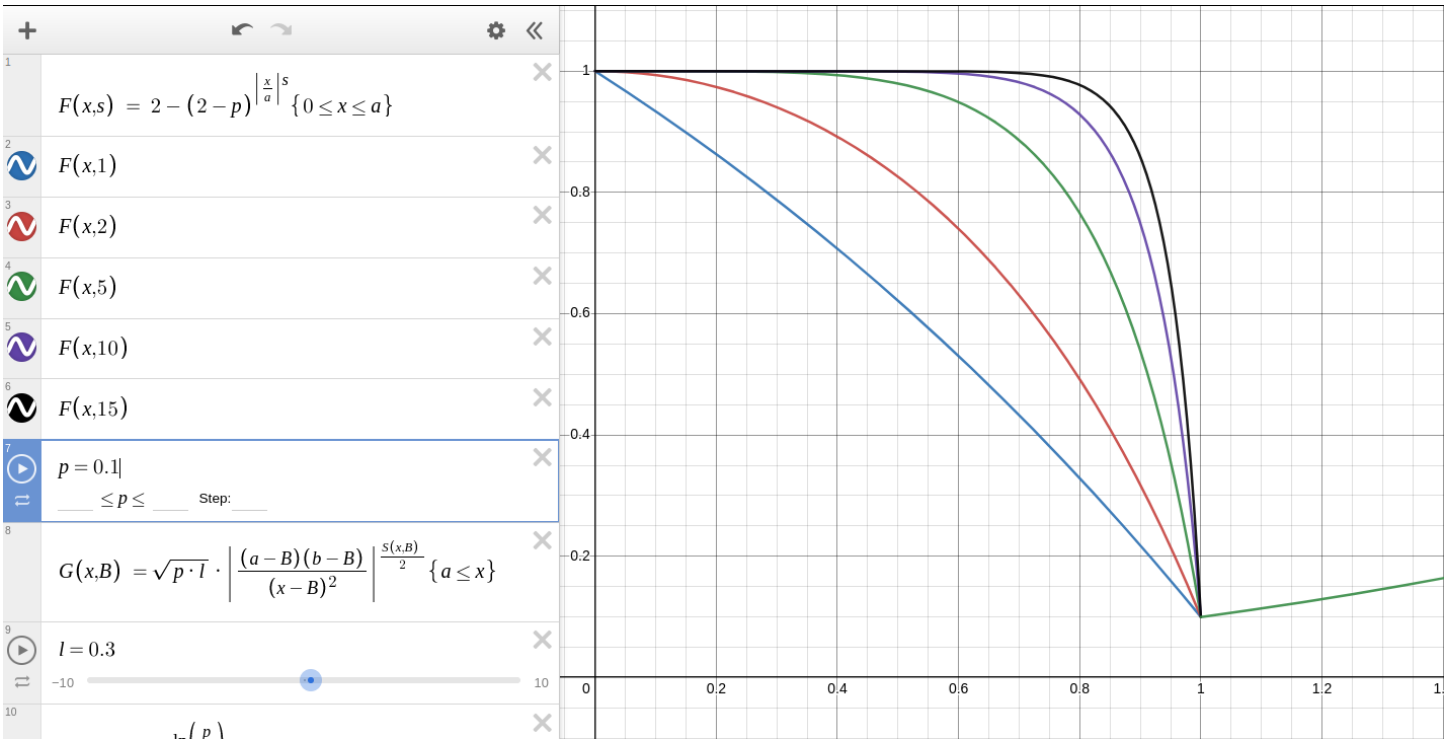


Figure 10: F(x) with different s values and for p= 0.1

- Employed when the creature is farther from the predator ($x \geq a$).
- p and l parameters shape the probability distribution.
- B parameter introduces a shift, customizing the response curve.
- Formula $\sqrt{p \cdot l} \cdot \left(\frac{(a-B)\cdot(b-B)}{(x-B)^2}\right)^{s/2}$ calculates probability with B consideration.
- s is in function of the variable B, $\frac{ln(\frac{p}{l})}{ln(\frac{b-B}{a-B})}$
- The original Model is $\frac{A}{(x-B)^s}$ with border conditions G(a) = p and F(b) = l

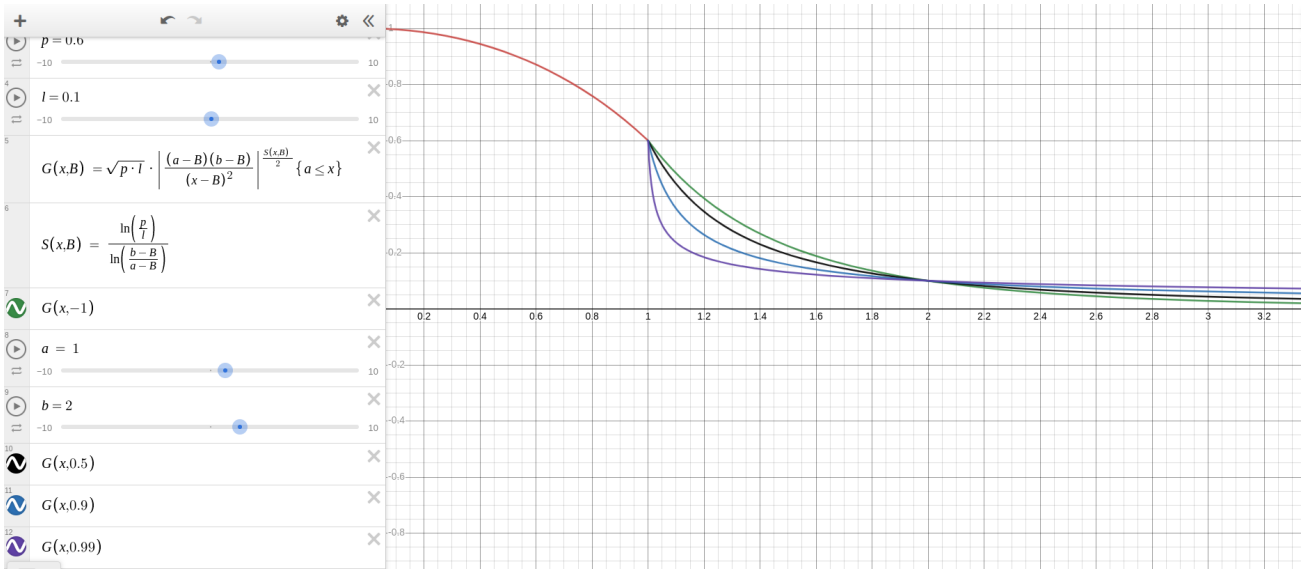


Figure 11: G(x) with different B values and for p= 0.6

5. Escape angle calculation:
   - Computed based on relative positions of the creature and predator, enabling evasive actions.

Conclusion:
The `Critique_Probability` function models predator-prey interaction. It uses F and G functions to calculate detection probability. The function maintains intuitive behavior: $F(0) = 1$ and $F(a) = p$ for proximity, $G(a) = p$ and $G(b) = l$ for distance. This approach creates a realistic transition across distances, enhancing the simulation's authenticity.

2 Inner Engine Parts (aquaengine)

2.1 Introduction

Welcome to the in-depth exploration of AquaEngine's inner engine parts. In this section, we'll dive into the intricate architecture and mechanics that power AquaEngine's core functionality, bringing aquatic life to digital existence.

This subsection unveils the foundation of internal components that drive AquaEngine's captivating aquatic simulation. We'll provide an overview of the engine's architecture, algorithms, and design considerations that combine science and creativity to craft a realistic underwater environment.

As we venture deeper, we'll dissect AquaEngine's physics simulations, revealing how buoyancy, hydrodynamics, and collision detection work together to create lifelike movements. We'll also explore how interactions among aquatic creatures are meticulously modeled, from predator-prey dynamics to schooling behaviors, adding depth and realism to the simulation.

2.2 Internal Components

Centrer_Gravity

The Centrer_Gravity function plays a crucial role in determining the movement behavior of each creature in the fish simulator. It is responsible for calculating the barycenter or center of mass of nearby creatures and influencing their attraction towards this central point.

Function Signature:

```
void Centrer_Gravity(Creature *creature, Cluster *cluster, double *t_X, double *t_Y,
                        double (*Evaluate)(double, double));
```

Parameters:

- `Creature *creature`: Pointer to the main creature around which the center of gravity is calculated.
- `Cluster *cluster`: Pointer to the cluster containing other creatures that contribute to the center of gravity calculation.
- `double *t_X`: Pointer to a variable that will store the calculated X-coordinate of the center of gravity.
- `double *t_Y`: Pointer to a variable that will store the calculated Y-coordinate of the center of gravity.
- `double (*Evaluate)(double, double)`: Pointer to an optional function that can be used to modify the contribution of individual creatures to the center of gravity calculation based on custom evaluation criteria.

Outputs:
None. The calculated X and Y coordinates of the center of gravity are stored in the variables pointed to by `t_X` and `t_Y`, respectively.

Purpose

- The purpose of the Centrer_Gravity function is to calculate the coordinates (X, Y) of the barycenter by considering the positions of other nearby creatures in the cluster. The barycenter acts as a virtual center of mass that creatures are attracted to, affecting their movement patterns and group behavior.
- The calculation of the barycenter involves evaluating the distances between the current creature and its neighboring creatures. To make this evaluation customizable and allow for various movement behaviors, the function accepts a function pointer called Evaluate.

Mathematical Function (Evaluate):

- The Evaluate function (represented by the function pointer Evaluate) describes how distances between the current creature and nearby creatures affect their influence on the barycenter. The attraction strength from nearby creatures is determined by this function, which depends on the distance between the current creature and its neighbors.
- Visualizing the mathematical function through graphs or charts can help users understand its impact on the barycenter and creature movement.

Barycenter and Attraction Behavior:

- The barycenter represents the center of mass of nearby creatures in the cluster. As the creatures move and interact with each other, the barycenter dynamically shifts based on their positions. Creatures tend to gravitate towards this central point, resulting in collective movement patterns and group cohesion.
- The Evaluate function allows users to customize the attraction behavior of creatures towards the barycenter. It can be adjusted to create different movement dynamics, such as stronger or weaker attraction forces, leading to various group formations and patterns.

Note:

- The Evaluate function can be modified to experiment with different movement behaviors in the fish simulator. Users can adjust the mathematical function to achieve their desired group dynamics and flocking patterns.
- The calculation of the barycenter and attraction behavior is essential in creating visually appealing and realistic movement animations of the creatures in the fish simulator.
- For further diagnostic and visualization of the gravity center of each instance, users can explore the debug mode section in "Inner Graphic Part." Debug mode provides valuable insights into the movement and behavior of individual creatures, aiding in the analysis and fine-tuning of the simulation.
- To gain a deeper understanding of the Centrer_Gravity function and the impact of the Evaluate function pointer on centering the creatures' movements, let's explain it in simpler terms. The barycenter, or center of mass, is a point within a system where the combined mass or concentration of an object can be considered to be concentrated. In the context of the fish simulator, it represents the virtual center point that pulls each creature towards itself, causing them to move collectively in a coordinated manner. The Evaluate function allows users to control the strength and behavior of this attraction.

getFictifCoordinates

The getFictifCoordinates function is responsible for converting real-world coordinates into fictitious coordinates within the non-Euclidean window space. This transformation is essential for accurately simulating the movements of creatures and ensuring they wrap around the screen seamlessly.

Function Signature:

```
double *getFictifCoordinates(double pivot,double guest,int axis_dimension);
```

Parameters:

- `double pivot:` The real-world coordinate of the pivot point.
- `double guest:` The real-world coordinate of the guest point.
- `int axis_dimension:` The size of the window in the specified axis (either X or Y).

Outputs:

The function returns a pointer to a dynamically allocated array containing two fictitious coordinates corresponding to the pivot and guest points, respectively.

Purpose:

The primary purpose of getFictifCoordinates is to handle translations across the non-Euclidean window space, where the screen's edges are connected.

Functionality:

The function allocates memory for the fictifCoordinates array to store the fictitious coordinates. It then sets the initial fictitious coordinates to be the same as the real-world coordinates. Based on the comparison between the distances of the guest point to the pivot point and the distances to the wraparound positions (axis_dimension + pivot and -axis_dimension + pivot), the function adjusts the guest point's fictitious coordinate to ensure a smooth wraparound effect. The fictitious coordinates represent the guest point's position after considering the wraparounds in the non-Euclidean window space.

Mathematical Insight:

The getFictifCoordinates function performs a simple mathematical comparison to determine whether it is more efficient for the guest point to wrap around the screen's edges to reach its fictitious position. By adjusting the guest point's position, the function ensures a seamless transition between the edges of the screen, maintaining the illusion of a connected non-Euclidean window space.

Note:

The getFictifCoordinates function is a vital component of the engine's non-Euclidean window space simulation. It enables creatures to move freely across the screen, making the aquatic world feel boundless and immersive. Proper handling of fictitious coordinates is critical to achieving smooth and natural movements for creatures in the simulation. Users can explore various screen sizes and window dimensions to observe the impact on creature behaviors and screen wrapping.

Folded_Space_Distance

The Folded_Space_Distance function plays a crucial role in calculating the distance between two points within the non-Euclidean window space. It takes into account the screen's wraparound effect to ensure accurate distance measurements across the simulated environment.

Function Signature:

```
double Folded_Space_Distance(double x1,double y1,double x2,double y2 );
```

Parameters:

- `double x1, y1`: The real-world coordinates of the first point.
- `double x2, y2`: The real-world coordinates of the second point.

Output

The function returns the distance between the two points, accounting for the wraparound effect within the non-Euclidean window space.

Purpose:

The primary purpose of Folded_Space_Distance is to calculate the distance between two points in the non-Euclidean window space, considering the wraparound effect due to the fictitious coordinate transformation.

Functionality:

The function uses getFictifCoordinates to convert the real-world coordinates of both points into fictitious coordinates. This ensures that the wraparound effect is correctly considered when calculating the distance. After obtaining the fictitious coordinates for both points, the function calculates the Euclidean distance between them in the non-Euclidean window space using the standard distance formula. The result represents the accurate distance between the two points, accounting for the screen's wraparound effect.

Mathematical Insight:

The calculation of the distance between two points in the non-Euclidean window space involves converting real-world coordinates into fictitious coordinates and then applying the standard distance formula. By using fictitious coordinates, the function ensures that the wraparound effect is correctly considered, providing precise distance measurements in the simulated aquatic world.

Note:

Folded_Space_Distance is a fundamental utility function in the engine, used throughout the simulation whenever distance calculations between points are required. To avoid redundant calculations, in some instances where fictitious coordinates are already available in the code, the distance is computed manually using the standard distance formula. This optimization minimizes the number of calls to Folded_Space_Distance and enhances overall performance. Proper handling of fictitious coordinates is essential for accurate distance measurements and smooth movement of creatures in the non-Euclidean window space.

2.3 Limitations

While AquaEngine offers a versatile framework for simulating underwater environments and aquatic life, it's important to acknowledge its limitations. Understanding these limitations can help developers make informed decisions and set realistic expectations when utilizing AquaEngine for their projects.

Computational Performance

Efficient computational performance is a fundamental aspect of AquaEngine's design to ensure real-time simulations of underwater environments. However, understanding the distribution of computational resources among different functions and their impact on overall performance is crucial for optimizing the engine's performance.

Profiling Results Profiling the AquaEngine simulation using the `gprof` tool provides insights into the time spent within various functions during the execution. The following key functions contribute significantly to the computational workload:

1. `getFictifCoordinates`: This function accounts for approximately 36.38% of the total execution time. It is utilized for calculating fictitious coordinates within the simulation and is a significant contributor to the overall computation.
2. `Centrer_Gravity`: Responsible for around 34.56% of the total execution time, this function computes the center of gravity of a creature within a cluster. It involves complex calculations and interactions among fish clusters, impacting the performance.
3. `Customized_Selection`: Contributing to approximately 23.64% of the total execution time, this function handles the selection of specific fish clusters based on certain criteria. The selection process involves a non-trivial amount of computation.

Optimization Considerations Based on the profiling results, developers can focus their optimization efforts on the functions that contribute most significantly to the execution time. Strategies to improve computational performance include:

- **Algorithmic Optimization:** Review the algorithms used within functions like `getFictifCoordinates` and `Centrer_Gravity` for potential optimizations. Simplifying calculations or employing more efficient algorithms can reduce execution time.
- **Parallelism:** Investigate opportunities for parallelizing computations, especially within functions that involve calculations across multiple fish clusters. Utilizing multi-threading or parallel processing can distribute the computational load.
- **Data Structures:** Optimize data structures and memory access patterns to reduce computational overhead. Efficient data storage and retrieval can lead to faster computations.
- **Caching:** Implement caching mechanisms to store intermediate results that are reused across iterations. This can minimize redundant calculations and improve overall efficiency.

- **Profile-Guided Optimization:** Use profiling tools like `gprof` to identify "hot spots" in your code. Profile-guided optimization involves refining code based on actual usage patterns, leading to targeted performance improvements.

Performance Monitoring Regularly monitoring the engine's performance as simulations become more complex can help identify potential bottlenecks and areas for optimization. It's essential to balance realism with performance, ensuring that AquaEngine delivers a visually appealing and responsive simulation experience.

By strategically addressing computational performance concerns and optimizing critical functions, developers can enhance the efficiency of AquaEngine simulations, enabling them to create captivating underwater environments while maintaining real-time responsiveness.

Simplified Physics

The AquaEngine can be likened to a skeletal structure, a framework that invites the exploration of mathematical functions and their relationship with natural phenomena. This analogy evokes the concept of a "bone without flesh," representing a foundational structure that awaits the layers of complexity that can be added to it. In this context, the AquaEngine's physics simulation serves as a starting point for understanding how mathematical formulations can describe and replicate natural behaviors.

Similar to how one studies human movement by examining the bones first, the AquaEngine focuses on the fundamental mathematical principles that underlie the movement and interactions of aquatic creatures. Just as layers of flesh can be added to bones to enhance strength and awareness, the AquaEngine can be developed further to incorporate more intricate details and nuanced behaviors, expanding its capabilities and realism.

Despite its simplicity, the AquaEngine's physics simulation carries a profound message—it showcases the elegance and power of mathematical descriptions in capturing the essence of natural processes. While the AquaEngine may have limitations, these limitations serve as a canvas for creative exploration, encouraging users to imagine, model, and experiment with the interplay of mathematical concepts and biological dynamics.

Hardware and Platform Dependency

The aquaengine is designed to operate primarily on a single core, making efficient use of the available processing power. However, as the number of fish within the simulation increases, there comes an opportunity to enhance performance through the utilization of multithreading within the Centrer_Gravity function. This function, responsible for calculating the center of gravity for clusters of fish, can benefit from multithreading to parallelize the computation, resulting in improved execution times on multicore processors.

In scenarios where the simulation demands high computational power, utilizing the NVIDIA CUDA toolkit (nvcc) can provide further performance optimizations. By offloading certain calculations to the GPU, which is optimized for parallel processing, the simulation can experience significant speedup, especially when handling larger numbers of fish or complex interactions.

It's important to note that while the current version of the aquaengine focuses on single-core execution, exploring these hardware and platform-dependent optimizations can provide a more responsive and efficient simulation experience, even on modern hardware architectures.

Customization and Extensibility

While AquaEngine is designed to be extensible and customizable, the scope of modifications might be limited by the underlying architecture. Developers seeking highly specialized behaviors or interactions might need to make significant code adjustments, potentially requiring a deep understanding of the engine's internal workings.

Learning Curve

Navigating and utilizing the diverse features and parameter configurations of `aquaengine` may initially present a learning curve. New developers approaching the engine should be prepared to invest some time in comprehending its intricate components, consulting its comprehensive documentation, and exploring best practices to harness its capabilities effectively.

An effective practice to ease this learning process is to preview and experiment with the mathematical functions and concepts used within `aquaengine` on external platforms. Web-based tools like WebDesmos can serve as invaluable aids in visualizing and understanding various mathematical functions. By using such resources, developers can gain a clearer grasp of the mathematical underpinnings that drive the engine's behavior.

Remember that mastery over `aquaengine` can unlock its full potential, allowing you to create simulations that align with your creative vision and achieve the desired simulation outcomes.

3 Inner Graphical Parts (aquaSDL)

3.1 Introduction

AquaSDL is a powerful and versatile graphics library designed to enhance the graphical capabilities of AquaEngine, a simulation and animation environment for aquatic creatures. AquaSDL serves as the graphical backbone, providing functionalities to render creatures, animate their movements, and manage interactions within the aquatic environment. The AquaSDL library introduces a set of drawing functions that facilitate the rendering of various graphical elements, such as fish, predators, and environmental components. These functions handle the positioning, rotation, and animation of sprites, offering a seamless and visually appealing experience.

3.2 AquaSDL Configuration

The AquaSDL Configuration section provides an overview of the configuration options available in AquaSDL and explains how to set up the graphical assets and parameters required for AquaEngine's graphics and animation.

1. **Sprite Sheets and Background:**

   AquaSDL relies on sprite sheets to render graphical elements such as fish, predators, and the background. The following variables are used for specifying the file paths to the sprite sheets and background image:

   - `where_fish`: Path to the sprite sheet containing fish animations.
   - `where_predateur`: Path to the sprite sheet containing predator animations.
   - `where_background`: Path to the background image.

   To configure AquaSDL for your project, set these variables with the appropriate file paths to the corresponding assets.

   ```
   extern char* where_fish;          // Path to the fish sprite sheet.
   extern char* where_predateur;     // Path to the predator sprite sheet.
   extern char* where_background;    // Path to the background image.
   ```

2. **Sprite Sheets Texture:**

   AquaSDL uses SDL_Texture objects to load and render the sprite sheets efficiently. The following variables represent the texture objects for the sprite sheets:

   - `spriteSheet_Fish`: SDL_Texture object for fish sprite sheet.
   - `spriteSheet_Predator`: SDL_Texture object for predator sprite sheet.

   Before rendering creatures, it's essential to load these textures from the corresponding sprite sheet surfaces.

   ```
   extern SDL_Texture* spriteSheet_Fish;       // Texture for fish sprite sheet.
   extern SDL_Texture* spriteSheet_Predator;   // Texture for predator sprite sheet.
   ```

3. **Fish Types:**

   AquaSDL categorizes fish into different types based on their animations. The following constants represent the fish types:

   - `BLEU_FISH`: Type for blue fish.
   - `BLEU_YELLOW`: Type for yellow fish.
   - `BLEU_RED`: Type for red fish.

   These types are used to access specific fish animations from the sprite sheet.

   ```
   #define BLEU_FISH 0
   #define BLEU_YELLOW 1
   #define BLEU_RED 7
   ```

4. **Debug Mode:**

   AquaSDL provides two debug mode variables to assist with debugging and visualizing creature behavior:

   - `low_debugmode`: If set to true, AquaSDL will display rectangles without sprites and show the center of gravity and how creatures cross borders.
   - `rotational_debugmode`: If set to true, AquaSDL will draw the rotational boundaries of the sprite rotation.

   Debug mode can be enabled or disabled as needed to visualize the internal behavior of creatures.

```
extern bool low_debugmode;           // Enable/disable low debug mode.
extern bool rotational_debugmode;    // Enable/disable rotational debug mode.
```

5. **Additional Configuration Options:**

   If there are other configuration options or variables in AquaSDL, this section will provide explanations and instructions for each of them.

   Please note that this section provides an overview of the configuration options available in AquaSDL. Detailed explanations, usage instructions, and code examples will be included in subsequent subsections to provide a comprehensive understanding of AquaSDL configuration.

3.3 Drawing Functions

getMaxDims

Function Signature:

```
int* getMaxDims(double width, double height, double angle);
```

Purpose:
This function calculates the maximum dimensions (width and height) of a rectangle after it has been rotated by a given angle. it's primarly used to create a new fictitious Headbox for the rotation

Parameters:

- `width`: The original width of the rectangle.
- `height`: The original height of the rectangle.
- `angle`: The angle of rotation in degrees.

Output:
The function returns a pointer to an integer array of size 2. The first element of the array represents the maximum width, and the second element represents the maximum height after the rotation.

Note:
The rotation is performed around the center of the rectangle.

Example:

```
double width = 100.0;
double height = 50.0;
double angle = 45.0;

int* maxDims = getMaxDims(width, height, angle);
printf("Maximum Width: %d, Maximum Height: %d\\n", maxDims[0], maxDims[1]);
free(maxDims);
```

Output:

```
Maximum Width: 106, Maximum Height: 106
```

Importance :
In the next video, we will witness in the `rotational_debugmode` the importance of `getMaxDims`, and its role in calculating the Rotated Sprite HeadBox. This calculation ensures a smooth transition between borders, illustrating the wrap-around property of the fictional aqua world.

- The red : Actual HeadBox of The instance
- The green : The Rotational Headbox of The instance

`https://youtu.be/i9_T1zXRlnM`
`https://youtu.be/zlSUhjirXl4`

DrawObject_In_debugmode

Function Signature:

```
void DrawObject_In_debugmode(SDL_Renderer *renderer, SDL_Rect MainRect, double *Center);
```

Parameters:

- `renderer`: Pointer to the SDL renderer.
- `MainRect`: `SDL_Rect` representing the main rectangle to be drawn.
- `Center`: Double pointer to an array containing the X and Y coordinates of the center of gravity for the main rectangle.

Output:
None

Purpose:

- The `DrawObject_In_debugmode` function is an essential part of the debug mode in AquaSDL. When `low_debugmode` is enabled, this function allows developers to visualize and inspect the behavior of rectangles within the simulation.
- The function helps in diagnosing any issues related to the movement, rotation, or positioning of objects on the screen.
- By displaying the center of gravity for each rectangle, developers can ensure that the gravitational interactions are working as expected.
- The function also visually demonstrates how rectangles manage to cross the screen borders when they exceed the window's boundaries. This visualization aids in understanding how the folding of space is handled in the simulation.

Notes:

- This function is meant for debugging and visualization purposes only.
- When using AquaSDL for debugging, developers can enable `low_debugmode` to obtain valuable insights into the behavior of fish and predators.
- For further diagnostic and visualization capabilities in the debug mode, developers can explore the debug modes provided in the "Inner Graphic Parts" section of AquaSDL.

DrawObject_In_Normal

Function Signature:

```
void DrawObject_In_Normal(SDL_Renderer *renderer, SDL_Rect *rect, SDL_Rect *fictif_rect,
                          double angle, SDL_Texture* spriteSheet_type, SDL_Rect SpriteRect);
```

Parameters:

- `renderer`: Pointer to the SDL renderer.
- `rect`: Pointer to an `SDL_Rect` representing the main rectangle to be drawn.
- `fictif_rect`: Pointer to an `SDL_Rect` representing the fictitious rectangle for positioning (can be NULL).
- `angle`: The angle of rotation for the object.
- `spriteSheet_type`: `SDL_Texture` containing the sprite sheet of the object.
- `SpriteRect`: `SDL_Rect` specifying the portion of the sprite sheet to be rendered.

Output: None

Purpose:

- The `DrawObject_In_Normal` function is responsible for drawing the main rectangle representing an object using the given sprite sheet and angle of rotation.
- This function is used when `low_debugmode` is disabled, and actual sprite rendering is required during the simulation.
- The `DrawObject_In_Normal` function allows the rendering of the main rectangle with the correct rotation angle and appearance based on the provided sprite sheet.
- The `fictif_rect` parameter is used when there is a need to display the object at a fictitious position. This helps in avoiding duplication of calculations already performed, enhancing performance when complex interactions require fictitious positioning.

Notes:

- This function is used for regular object rendering during the simulation when `low_debugmode` is disabled.
- When `low_debugmode` is enabled, the `DrawObject_In_debugmode` function is used for visualization and diagnostic purposes.
- The `drawOutlineRectangles` function is called internally to draw the outline rectangles when objects exceed the window boundaries due to folding of space.

drawOutlineRectangles

Function Signature:

```
void drawOutlineRectangles(SDL_Renderer *renderer, double alpha, SDL_Rect *rect,
                           SDL_Rect **Returned_fictif_rect);
```

Parameters:

- `renderer`: Pointer to the SDL renderer.
- `alpha`: The angle of rotation for the object (used for rotational debugging).
- `rect`: Pointer to an `SDL_Rect` representing the main rectangle to draw.
- `Returned_fictif_rect`: Pointer to an `SDL_Rect` pointer that will be set to the fictitious rectangle created during the function (can be NULL).

Output:

None

Purpose:

- The `drawOutlineRectangles` function is used to draw the outline rectangles of an object, both in its normal position and its fictitious position (if needed).
- This function is utilized for debugging and visualization purposes, especially when `rotational_debugmode` is enabled.
- The function calculates the maximum dimensions (width and height) of the rotated object, considering the provided rotation angle `alpha`.
- It draws the outline rectangle in the normal position and, if required, the fictitious position (using the `drawRotatedRect` function).
- The function creates a fictitious rectangle, representing the object's bounds when placed at its fictitious position, which can be returned to the calling function through the `Returned_fictif_rect` parameter.

Notes:

- This function is used to visualize the rotational boundaries of objects during the simulation.
- The `drawRotatedRect` function is called internally to draw the rotated rectangles based on the calculated maximum dimensions.
- When `rotational_debugmode` is disabled, this function does not have any visible effect on the simulation.

3.4 Animation Functions

AnimatePredator

Function Signature:

```
void AnimatePredator(Cluster *cluster, void *renderer, Creature *predator);
```

Parameters:

- `cluster`: Pointer to the Cluster structure containing fish and predator instances.
- `renderer`: Pointer to the SDL renderer (used for graphical rendering).
- `predator`: Pointer to an array of Creature structures representing predator instances.

Output:

None

Purpose:

- The `AnimatePredator` function is responsible for animating and drawing predator instances on the screen.
- If graphical mode is enabled (`GRAPHIC_MODE_FOR_THE_ENGINE`), the function animates the predators using sprite sheets and draws them in their current positions.
- When `low_debugmode` is disabled, the function draws the animated predators using sprite sheets. Otherwise, it displays simple rectangles with the gravity centers of the predators.

- The function also calls the `mv_object` function to move the predator instances based on their interactions with other creatures in the cluster.

Notes:

- This function is used in the graphical mode of the AquaEngine to display and animate predator instances.
- If `low_debugmode` is enabled, the predators are displayed as rectangles, and their gravity centers are marked for visualization purposes.
- The `mv_object` function is called to update the positions of the predator instances based on their interactions with other creatures in the cluster.

AnimateFish

Function Signature:

```
void AnimateFish(Cluster *cluster, void *renderer, Creature *predator);
```

Parameters:

- `cluster`: Pointer to the Cluster structure containing fish and predator instances.
- `renderer`: Pointer to the SDL renderer (used for graphical rendering).
- `predator`: Pointer to an array of Creature structures representing predator instances.

Output:

```
None
```

Purpose:

- The `AnimateFish` function is responsible for animating and drawing fish instances on the screen.
- It iterates over the clusters of fish instances and animates each fish in the cluster.
- If graphical mode is enabled (`GRAPHIC_MODE_FOR_THE_ENGINE`), the function animates the fish using sprite sheets and draws them in their current positions.
- When `low_debugmode` is disabled, the function draws the animated fish using sprite sheets. Otherwise, it displays simple rectangles with the gravity centers of the fish.
- The function also calls the `mv_object` function to move the fish instances based on their interactions with predators and other fish in the cluster.

Notes:

- This function is used in the graphical mode of the AquaEngine to display and animate fish instances.
- If `low_debugmode` is enabled, the fish are displayed as rectangles, and their gravity centers are marked for visualization purposes.
- The `mv_object` function is called to update the positions of the fish instances based on their interactions with predators and other fish in the cluster.

3.5 Debug Mode

The Debug Mode in AquaSDL serves as a valuable tool for visualizing and understanding the behavior of the creatures and their interactions within the AquaEngine. It offers two distinct debug modes, namely `low_debugmode` and `rotational_debugmode`, each providing different visualizations to aid in debugging and testing.

1. **low_debugmode:**

   - **Description:** When `low_debugmode` is enabled, the AquaEngine displays rectangles for creatures without loading their associated sprites. This mode is particularly useful for checking the position, size, and collisions of creatures without the added complexity of sprites.
   - **Usage:** To enable `low_debugmode`, set the variable `low_debugmode` to `true` in the AquaSDL configuration section. To disable it, set the variable to `false`.
   - **Effects:** In `low_debugmode`, creatures are represented as colored rectangles. Their centers of gravity are displayed with a green line, and they are shown crossing the borders of the environment.
   - **Video Preview:** `https://youtu.be/4YlY-iCnKmM`

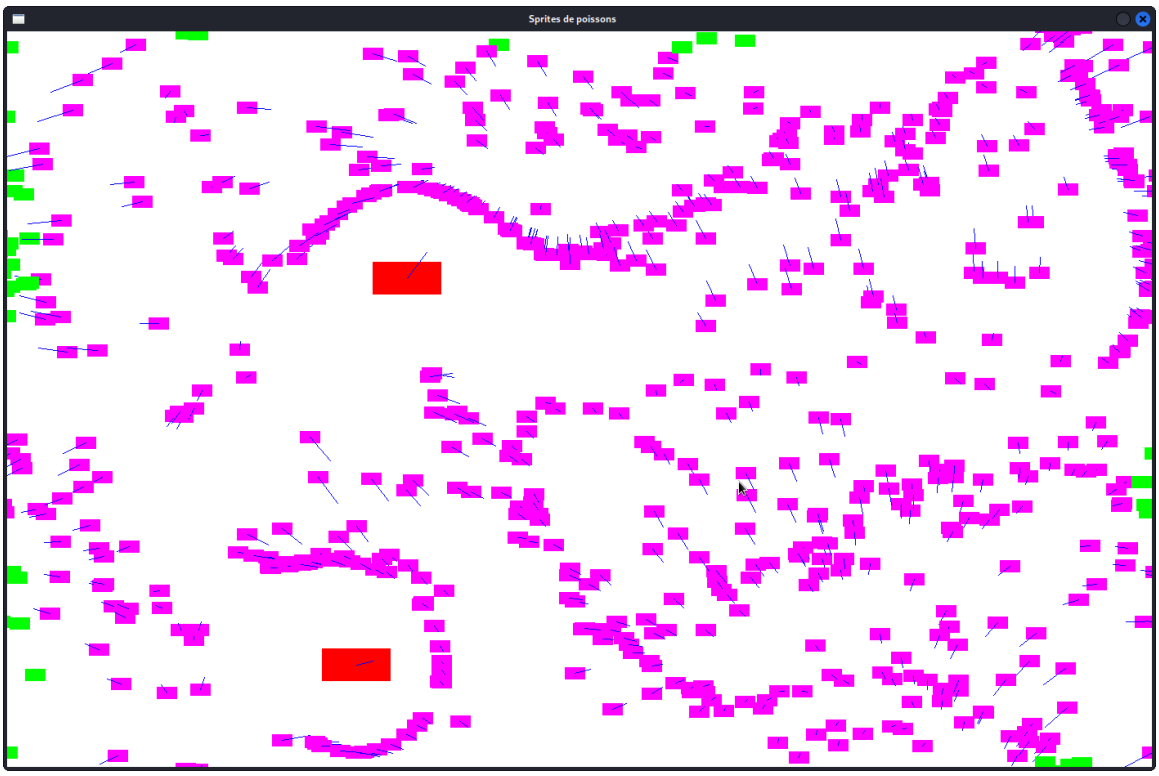


Figure 12: Preview of low_debugmode

2. **rotational_debugmode:**

   - **Description:** When `rotational_debugmode` is enabled, the AquaEngine displays the rotational boundaries of the sprite rotation for creatures. This mode helps visualize how the sprites are rotated around their center points.
   - **Usage:** To enable `rotational_debugmode`, set the variable `rotational_debugmode` to `true` in the AquaSDL configuration section. To disable it, set the variable to `false`.
   - **Effects:** In `rotational_debugmode`, the main rectangle of each creature instance and its three mirrors (if they exist) are drawn with green rectangles. This visualization illustrates how sprites penetrate boundaries and how they are shown on the other side of the screen. The `drawOutlineRectangles` function is crucial for achieving smooth translations and providing the `Returned_fictif_rect`.
   - **Video Preview:** `https://youtu.be/aVm3NQ16Z68`

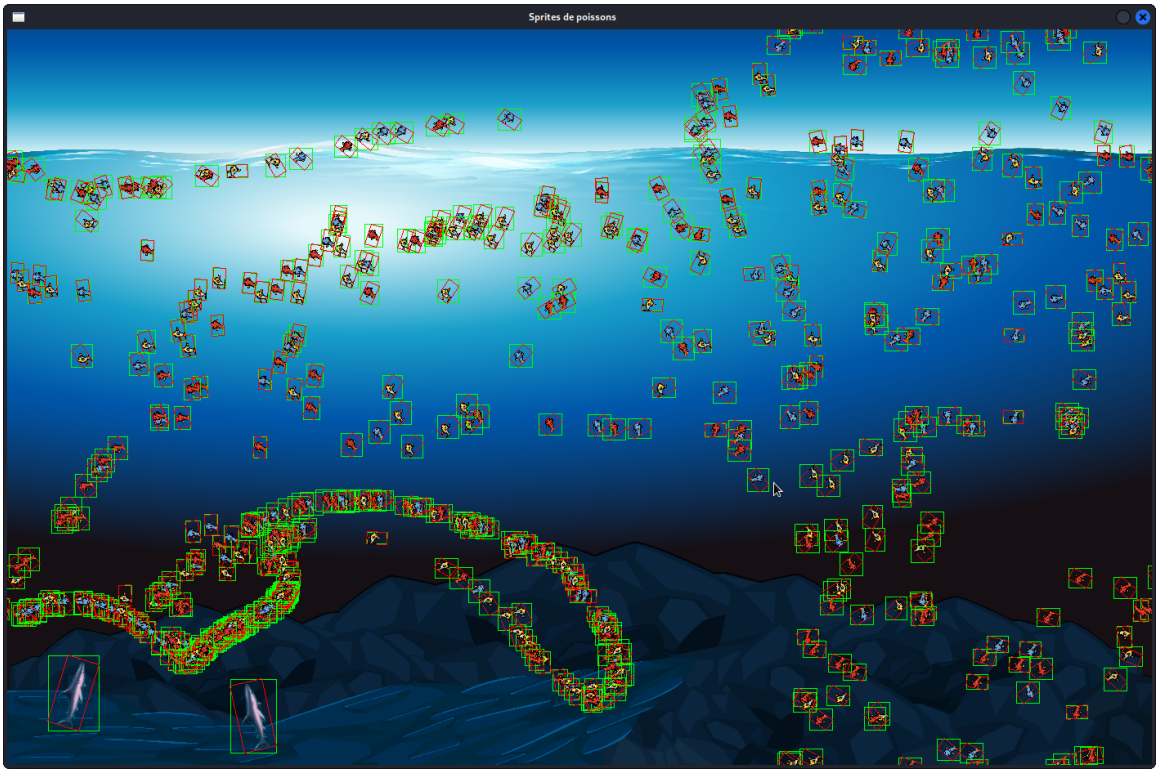


Figure 13: Preview of rotational_debugmode

Note: Debug mode is primarily intended for debugging and development purposes. It is recommended to disable debug mode when running AquaEngine in production or for actual use, as it may affect the visual appearance and performance.

The `low_debugmode` and `rotational_debugmode` are independent of each other, allowing you to choose one or both modes based on your debugging needs.

3.6 Additional Notes

While AquaSDL leverages the capabilities of SDL2 for graphics and animation, it's essential to be aware of some of the limitations that SDL2 might have when developing certain graphical applications. SDL2 is a powerful and straightforward library, but it has its boundaries.

Limitations of SDL2:

1. **Hardware Acceleration:** SDL2 provides hardware-accelerated rendering for 2D graphics, but it might not be as efficient as specialized graphics libraries like OpenGL for handling complex graphical tasks.
2. **Shader Support:** SDL2 does not have built-in support for shaders, which are essential for implementing advanced graphical effects in modern applications.
3. **Graphics Complexity:** While SDL2 is well-suited for simple 2D graphics and basic animation needs, it might face challenges when dealing with highly complex or demanding graphical scenarios.

Integrating OpenGL: To overcome some of the limitations of SDL2 and tap into the full potential of modern graphics capabilities, integrating OpenGL into your AquaSDL project can be highly advantageous. OpenGL is an open-standard, cross-platform graphics API that offers advanced features and performance for rendering 2D and 3D graphics.

By integrating OpenGL with AquaSDL, you can achieve:

1. **Enhanced Graphics:** OpenGL allows for more sophisticated and realistic graphical effects, including advanced lighting, shading, and post-processing effects.

2. **Shader Support:** OpenGL provides extensive shader support, enabling you to create custom shaders for more complex and dynamic visual effects.

3. **Cross-Platform Compatibility:** OpenGL works seamlessly across different platforms, ensuring consistent graphical performance on various operating systems.

Note: To learn more about SDL2 and OpenGL, we encourage readers to explore the official documentation and online resources for both libraries. Additionally, consider the specific requirements and complexity of your graphical project before deciding on the integration of OpenGL.

3.7 Conclusion

AquaSDL plays a crucial role in AquaEngine, elevating the visual experience of the aquatic ecosystem. By providing an intuitive interface for rendering and animating creatures, AquaSDL empowers users to create captivating and realistic simulations. Its flexible configuration options and debug modes offer a seamless experience, while the integration with SDL ensures efficient graphics processing. For users looking to explore the world of aquatic simulations and animations, AquaSDL is a valuable asset.