CMP404 – 1703221

Applied Game Technologies

Keywords:

Boid – An entity that adheres to simple rules exhibits emergent flocking behaviour when in a group.

Introduction:

This project was to make a game for the PlayStation VITA, making use of its camera and the university's Game Education Framework (GEF) to implement Augmented Reality (AR) functionality.

The idea for the game was to implement Craig Reynold's flocking algorithm and have the user be able to interact with the "Boids" in real-time. The original concept for the game was to have farm animals walking around on the table in front of the user and the user to herd them into a pen using a marker. However, as the prototype evolved, the idea to make the Boids use 3 dimensions (and not be limited to the 2-dimensional plane on the tabletop) became more appealing. The Boids became fish, which became the concept for the final game, "aquARium".

In the game, blue fish swim around in front of the user. The user can rotate the camera around the fish freely and observe their behaviour. Orange "paintballs" can be fired at the fish from the camera. When these collide the fish change colour from blue to orange. The aim of the game is to change the colour of all the fish to orange.

How to play:

The D-pad is used to navigate the menus in the game.

"CROSS" is used to select the option in the menu.

"TRIANGLE" is used to return to the main menu.

In game, "R2" is to fire a paintball.

In game, pressing "select" allows the user to navigate and manipulate the forces determining the boids' behaviour. Up and down on the D-pad is used to highlight which variable to manipulate. Left and right on the D-pad increments and decrements the scale of the selected force.

For optimal flexibility of camera movement, the markers should be set up in a corner like so:

When first entering the game, it is required that the user allows all 5 markers to be detected simultaneously. This is to allow for transformation matrices to be calculated between markers 2 – 5 and marker 1. This is needed for smooth transitions between markers when the current marker is occluded from view. Once this is calculated the game begins.

Application Design:

(UML diagram on final page)

ARApp:

This is the main class for the application. It extends from the GEF Application class and overrides its Init, Update, Render and CleanUp functions. Instances of GEF helper classes; InputManager, AudioManager, SpriteRenderer and Renerer3D are created on the heap and pointers to these are passed to a StateMachine class which is created here.

The Update and Render functions get access to the current state the application is in by calling StateMachine's GetState function. The Update function calls the HandleInput and Update functions of the current state. ARApp's Render function calls the Render function of the current state.

Fonts and audio are loaded here as they are used by all states.

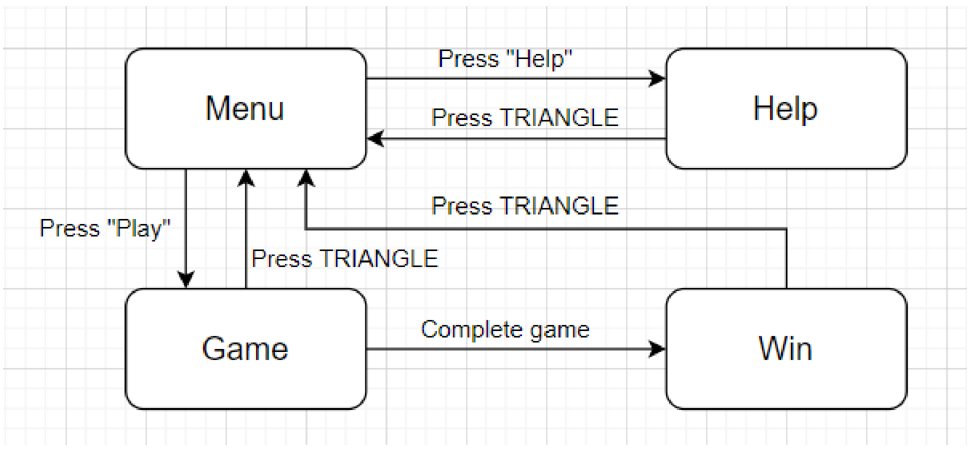
StateMachine:

This class handles the switching of states in the game. It creates instances of MenuState, GameState, OptionsState and WinState. These all inherit from a base class, State. It has an enumerator to indicate the state the application is currently in. It has a pointer of type State that points to the current child State object. This is making use of Polymorphism.

It has a SetState function that takes an enumerator as an argument that indicates the desired state to change to. This function calls the Release function of the current state, which essentially resets the state to the condition that would be expected upon entering the state, sets the pointer to the memory location of the desired state and calls the Init function of the new state.

Clean up of objects declared on the heap, to avoid memory leaks, is handled in the destructor of the states. States are not destroyed when switching.

State Machine Diagram:



State:

This is an abstract class that has pure virtual functions called Init, HandleInput, Update, Render and Release. Its child classes override these functions and are called in the main game loop. It also stores pointers to the various GEF helper classes that can be accessed by all the child classes. A Frames Per Second variable is declared here as this is needed for all states. The orthographic, view and projection matrices are declared here as they are also used by every state. This might not be the best approach for data locality but the effect on performance would be negligible. This object-oriented approach makes the code more readable and more easily scalable.

MenuState:

This class handles the title screen. It is what the user is greeted with when loading the game. It is simplistic in appearance, but the use of animated 3D models adds a lot to it. This is achieved by using the GEF Renderer3D class in the same way as in the main game. Care had to be taken to orient and scale the models as by default the model was oriented on the Z axis. The class makes use of the SpriteRenderer class to display the background image. There is a separate sprite for the button that indicates which option is selected. The logic for navigating the menu is very simple. There are three options, Play, Help and Quit. Pressing Quit simply exit the program.

HelpState:

This is a very simplistic state that simply displays text explaining how to play the game and the controls.

WinState:

This is a simplistic state that displays a message to the user indicating that they have completed the game. 3 dimensional models are utilized in this state, like in Menu.

GameState:

This class handles the actual gameplay. It makes use of various SONY libraries to give it functionality for tracking markers with the VITA’s camera. These libraries return transformation matrices describing the position, rotation and scale of the markers. This information allows for 3 dimensional geometry to be rendered in relation to real world positions, which allows for Augmented Reality.

A texture and sprite are required to render the image from the camera. This is updated every frame. The image has to be scaled to the resolution of the VITA’s camera.

A variable for the number of markers is declared. This determines how many markers the application will use. This is useful for different environments users might be playing the game in. Ideally 5 markers should be set up on a corner, 2 on the x plane, 2 on the y plane and one on the z plane (the table top). This will not always be possible however, so this is flexible. In retrospect this variable should be exposed to the user.

A Boolean for indicating when there is at least one marker is declared. This is used for determining when to render geometry. If geometry was rendered regardless, it would not know where it should be rendered when no markers are in view. It would be rendered in last position it was told and would move with the camera. This would completely break emersion. Choosing to have the geometry just not render was a lot less jarring from a gameplay perspective.

Variables to keep track of the ID and transformation matrices of the markers are declared. These are needed for the logic for switching the markers the game objects are anchored to.

A Boolean that states if the game is in its “calibration” stage was needed. This “calibration” stage is a required stage at the start of the game where all markers need to be detected by the camera in the same frame. This is so that transformation matrices can be calculated describing the transformation from each marker to the “main” marker. These are stored locally in this class.

A vector of Boids is populated, ready to be updated and rendered.

Update is where all the logic for switching markers is handled. The transform for the marker mesh instance is updated to the current marker transform being used. Every frame, the parent and offset transformation matrices are updated for each fish. All the fishes’ Update and Animate functions are called here also. Paintballs Update functions are called and sphere-sphere collision code is executed between all of the paintballs and fish. This section could be re-designed as there is a lot of looping here. If a collision is found the fish changes colour and the paintball is marked as not alive. It is no longer rendered and included in collision checks.

The geometry is then rendered on top of the camera sprite in the Render function.

In Release, variables are reset so that when re-visiting GameState the game is reset.

GameObject:

GameObject is a simple class that contains four matrices. A local, parent, offset, and world transform. The local transform describes the transform of the game object relative to its parent multiplied by an offset transform that describes the transform from the parent transform to the main marker transform. The world transform is the product of all of these matrices and describes its position and orientation in world space.

Boid:

Boid inherits from Game object. It is here where the local transform of GameObject is calculated. The position in local space is determined by Craig Reynold’s flocking algorithm. Initially the boids are spawned in random positions in a given environment. They each have a position, velocity and acceleration, stored as Vector4s.

The acceleration and velocity of the boids are determined by “steering behaviours”. These behaviours come from three main rules that each boid adheres to. These are cohesion, alignment and separation. With cohesion each boid steers towards the centre of mass of all the boids in its perception radius. With alignment each boid steers towards the average direction vector of all the boids in its perception radius. With separation, a steering vector is calculated that is the clamped sum of the vectors pointing away from each other boid in its perception radius, scaled by the distance away from it. This gives the effect of the boid steering more dramatically away from the boids it is most close to.

There is also a strong force added to the boid when hitting the boundary of its environment, steering it away from the wall.

When a boid gets too close to the camera there is also a strong force added in the direction directly away from the camera to simulate evasion.

All of these forces are added to the boid’s acceleration. This is calculated each frame. This acceleration is added to velocity, which is added to position, to give smooth movement.

Weighting for each of these behaviours can be edited at runtime from the game to allow the user to experiment with how they affect the boids’ behaviour.

Some additional vector math functions were created to allow extra functionality like clamping a vector between two magnitudes.

A translation matrix was constructed with the position of the boid. A rotation matrix was constructed using the pitch and yaw of the velocity vector. The product of the matrices was the local transform for the Boid and GameObject.

Fish:

Fish extends from Boid. It has a body and tail transform that is applied on top of the local transform of the game object. A hierarchical matrix transform approach was used to achieve basic animation. This involved translating each section of the model to its centre of rotation, applying a rotation and then translating it back. This was handled in the Animate function.

Paintball:

Paintball extends from GameObject. It simply has a position and velocity. When it is initialized, velocity is added to position until it hits a fish or it goes out of range.

There is only one mesh instance used for any one fish, marker or paintball at a time. The mesh and transform is changed on that mesh instance before every render. This was an attempt to save on memory and to improve data locality to increase performance.

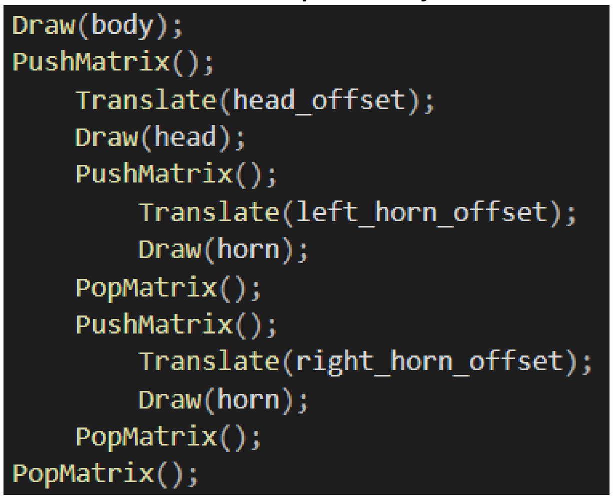
Critical Reflection:

Cross platform:

At the beginning of the semester, we had limited access to the university due to COVID-19 restrictions, which meant limited access to the PlayStation VITA's. This posed a problem for development. I attempted to address this problem by making a virtual environment for me to work from home. I made virtual markers that I could move around and a camera to navigate the environment. This way I was able to create objects and transform them in relation to the markers like I would be doing on the VITA. I feel that this was a good use of time as I was able to get practice working with matrices and develop game mechanics from home, however a lot of time spent in university was spent re-creating what I had done on the new platform. There were issues I ran into to do with scaling and using different co-ordinate systems that did confuse me when attempting to recreate it with the SONY system.

Models:

In earlier iterations of the project, the animals were made up entirely using cuboids from the provided primitive builder. These were translated into their proper positions using translation matrices. I created PushMatrix and PopMatrix functions that would add and remove matrices respectively to a stack data structure. I then had Scale, Rotate and Translate functions that would manipulate the last matrix on the stack. When rendering the mesh instances, I would transform them by the product of all the matrices on the stack. This allowed me to place objects relative to their parents.



```
Draw(body);
PushMatrix();
    Translate(head_offset);
    Draw(head);
    PushMatrix();
        Translate(left_horn_offset);
        Draw(horn);
    PopMatrix();
    PushMatrix();
        Translate(right_horn_offset);
        Draw(horn);
    PopMatrix();
PopMatrix();
```

Building the animal model with this approach was very expensive however and limited the number of Boids I was able to render whilst maintaining 30 frames per second. I found that creating models in blender and using the model loader achieved higher frames per second. This removed the need for this approach, so I decided to scrap it and instead just multiply the transformation matrices together step by step. By removing this abstraction, I was able to more clearly see the order of transformations.

Markers:

I wanted the user to be able to have as much freedom of movement of the camera as possible without having to worry about losing track of markers, which would break emersion. I had to decide how I would handle the case where the marker the game object was using got occluded from view. I decided a good way of doing this would be to calculate the transform between the marker that had just been occluded and a new marker that was not occluded, then set the game object’s parent transform to be the new marker’s transform multiplied by the offset transform. This would keep the game object in the same relative position.

This technique worked, however after a few transitions it became obvious that something was wrong as the game object would warp from its original orientation. I found a good solve to this was to calculate the offsets for each of the markers relative to the main marker at the beginning of the application and store them as constant matrices that would be multiplied by the current marker’s matrix. The only downside being a small amount of set up before the game.

I used the sprites I created for the markers in the PC application to give the user feedback about which marker was currently being used as an anchor for the game objects which was also very useful for visual debugging.

Limitations and areas for improvement:

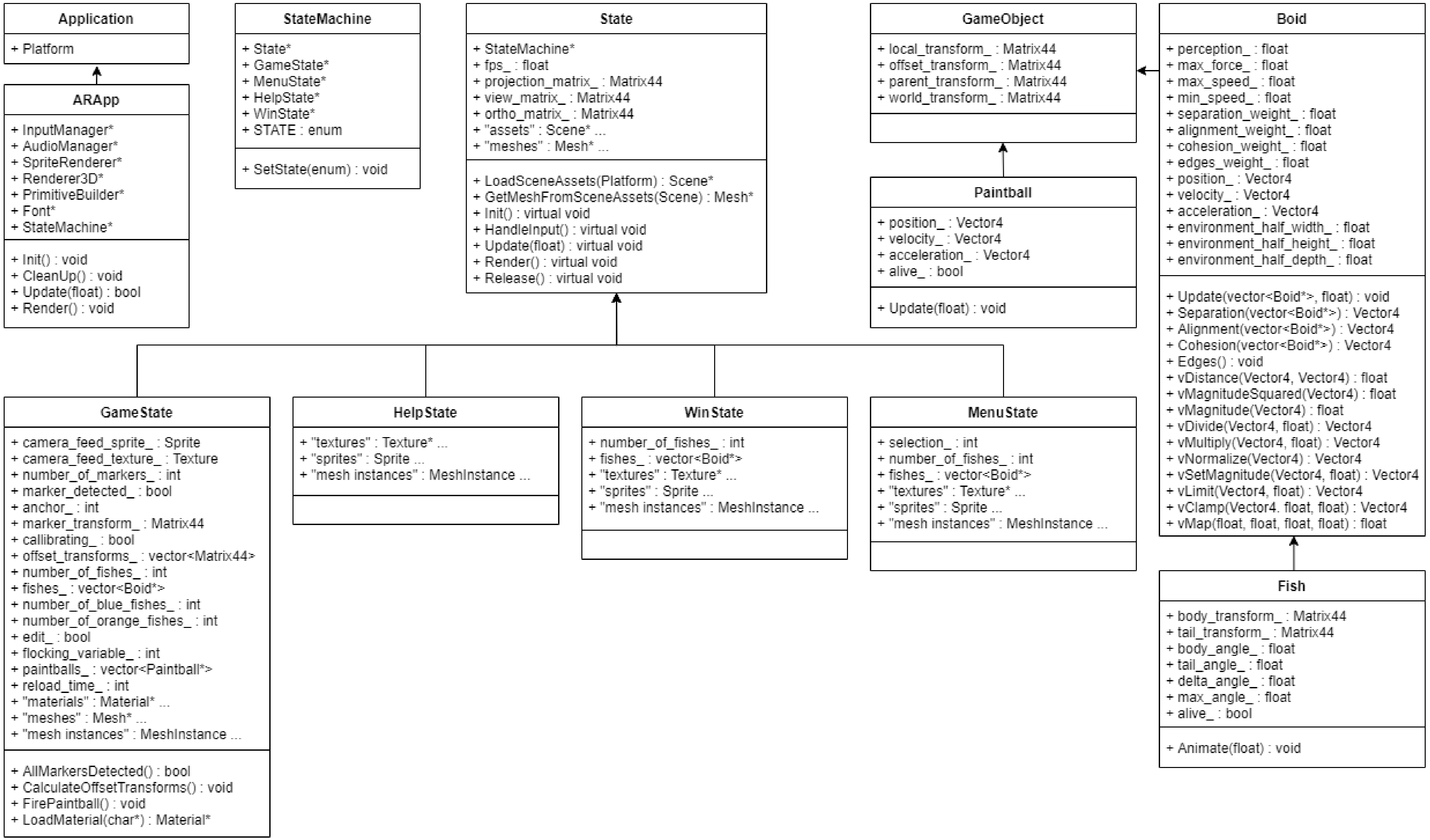
In my early game design I wanted the boids to run from a marker that was being moved by the player. This posed a problem as the player needed at least one hand to hold the PlayStation VITA. They could then not access all of the buttons on the console. I also found that ideally I would want the player to have be able to move 2 of the markers, one with each hand, in order to effectively control the boids’ movement and herd them into a pen. What would have been ideal would have been to have the console hands free.

Head Mounted Displays (HMD) have been around for a long time. The first Head-Mounted Display was created by Ivan Sutherland in 1968. He called it the “Sword of Damocles”. It was extremely primitive, very large, made of metal and was attached to the ceiling. It was not exactly practical public use.

Much improvement has been made since then offering Augmented reality but the hardware required still makes the headsets too big and awkward for every day use, limiting its appeal. For gaming however, the use of head mounted displays could allow for many innovations for computer games using this technology.

References:

Jerald, J. (2014), The Battle for Head--Mounted Displays.


Application
+ Platform
ARApp
+ InputManager*
+ AudioManager*
+ SpriteRenderer*
+ Renderer3D*
+ PrimitiveBuilder*
+ Font*
+ StateMachine*
+ Init() : void
+ CleanUp() : void
+ Update(float) : bool
+ Render() : void
StateMachine
+ State*
+ GameState*
+ MenuState*
+ HelpState*
+ WinState*
+ STATE : enum
+ SetState(enum) : void
State
+ StateMachine*
+ fps_ : float
+ projection_matrix_ : Matrix44
+ view_matrix_ : Matrix44
+ ortho_matrix_ : Matrix44
+ "assets" : Scene* ...
+ "meshes" : Mesh* ...
+ LoadSceneAssets(Platform) : Scene*
+ GetMeshFromSceneAssets(Scene) : Mesh*
+ Init() : virtual void
+ HandleInput() : virtual void
+ Update(float) : virtual void
+ Render() : virtual void
+ Release() : virtual void
GameObject
+ local_transform_ : Matrix44
+ offset_transform_ : Matrix44
+ parent_transform_ : Matrix44
+ world_transform_ : Matrix44
Paintball
+ position_ : Vector4
+ velocity_ : Vector4
+ acceleration_ : Vector4
+ alive_ : bool
+ Update(float) : void
Boid
+ perception_ : float
+ max_force_ : float
+ max_speed_ : float
+ min_speed_ : float
+ separation_weight_ : float
+ alignment_weight_ : float
+ cohesion_weight_ : float
+ edges_weight_ : float
+ position_ : Vector4
+ velocity_ : Vector4
+ acceleration_ : Vector4
+ environment_half_width_ : float
+ environment_half_height_ : float
+ environment_half_depth_ : float
+ Update(vector<Boid*>, float) : void
+ Separation(vector<Boid*>) : Vector4
+ Alignment(vector<Boid*>) : Vector4
+ Cohesion(vector<Boid*>) : Vector4
+ Edges() : void
+ vDistance(Vector4, Vector4) : float
+ vMagnitudeSquared(Vector4) : float
+ vMagnitude(Vector4) : float
+ vDivide(Vector4, float) : Vector4
+ vMultiply(Vector4, float) : Vector4
+ vNormalize(Vector4) : Vector4
+ vSetMagnitude(Vector4, float) : Vector4
+ vLimit(Vector4, float) : Vector4
+ vClamp(Vector4. float, float) : Vector4
+ vMap(float, float, float, float) : float
Fish
+ body_transform_ : Matrix44
+ tail_transform_ : Matrix44
+ body_angle_ : float
+ tail_angle_ : float
+ delta_angle_ : float
+ max_angle_ : float
+ alive_ : bool
+ Animate(float) : void
GameState
+ camera_feed_sprite_ : Sprite
+ camera_feed_texture_ : Texture
+ number_of_markers_ : int
+ marker_detected_ : bool
+ anchor_ : int
+ marker_transform_ : Matrix44
+ callibrating_ : bool
+ offset_transforms_ : vector<Matrix44>
+ number_of_fishes_ : int
+ fishes_ : vector<Boid*>
+ number_of_blue_fishes_ : int
+ number_of_orange_fishes_ : int
+ edit_ : bool
+ flocking_variable_ : int
+ paintballs_ : vector<Paintball*>
+ reload_time_ : int
+ "materials" : Material* ...
+ "meshes" : Mesh* ...
+ "mesh instances" : MeshInstance ...
+ AllMarkersDetected() : bool
+ CalculateOffsetTransforms() : void
+ FirePaintball() : void
+ LoadMaterial(char*) : Material*
HelpState
+ "textures" : Texture* ...
+ "sprites" : Sprite ...
+ "mesh instances" : MeshInstance ...
WinState
+ number_of_fishes_ : int
+ fishes_ : vector<Boid*>
+ "textures" : Texture* ...
+ "sprites" : Sprite ...
+ "mesh instances" : MeshInstance ...
MenuState
+ selection_ : int
+ number_of_fishes_ : int
+ fishes_ : vector<Boid*>
+ "textures" : Texture* ...
+ "sprites" : Sprite ...
+ "mesh instances" : MeshInstance ...